Debugging Aspect-Enabled Programs

Marc Eaddy[1], Alfred Aho[1], Weiping Hu[2], Paddy McDonald[2], Julian Burger[2]

[1] Department of Computer Science
Columbia University
New York, NY 10027
{eaddy, aho}@cs.columbia.edu

[2] Microsoft Corporation
One Microsoft Way
Redmond, WA 98052
{weipingh, paddymcd, julianbu}@microsoft.com

Abstract. The ability to debug programs composed using aspect-oriented programming (AOP) techniques is critical to the adoption of AOP. Nevertheless, many AOP systems lack adequate support for debugging, making it difficult to diagnose faults and understand the program's composition and control flow. We present an *AOP debug model* that characterizes AOP-specific program composition techniques and AOP-specific program behaviors, and relates them to the AOP-specific faults they induce. We specify debugging criteria that we feel all AOP systems should support and compare how several AOP systems measure up to this ideal.

We explain why AOP composition techniques, particularly dynamic and binary weaving, hinder source-level debugging, and how results from related research on debugging optimized code help solve the problem. We also present Wicca, the first dynamic AOP system to support full source-level debugging. We demonstrate how Wicca's powerful interactive debugging features allow a programmer to quickly diagnose faults in the base program behavior or AOP-specific behavior.

1 Introduction

We use the term *debuggability* to mean the ability to diagnose faults in a software system, and to improve comprehension of a system, *by monitoring the execution of the system*. Many debugging techniques exist, including source-level debugging, *printf*-style debugging, assertions, tracing, logging, and runtime visualization.

The ability to debug aspect-enabled programs is important for many reasons. The interaction of aspects with a system introduces new fault types and complicates fault resolution [2]. Programmers rely on debugging to diagnose these faults and perform post-mortem analyses. Debugging is also an important tool for program comprehension. Aspect functionality can drastically change the behavior and control flow of the base program, leading to unexpected behavior [2] and resulting in the same complexity that multi-threaded programs are notorious for. Debugging provides a way to demystify these intricacies and better understand the composed program.

Aspect-oriented programming (AOP) [28] is still an emerging field with many different techniques for aspect specification, composition, and integration. Along with tool support, debugging support serves as an indicator of AOP maturity [17, 32]. Commercial software developers are hesitant to adopt aspect-oriented software development practices or ship AOP-enabled products that are difficult to debug and service [2, 17, 24, 25].

Debugging is no substitute for *aspect visualization* [17] and testing. Indeed they are complementary: aspect visualization provides the ability to predict aspect behavior; testing provides a process for automatically detecting anomalies; and debugging provides a way to manually detect, diagnose, and fix anomalies and to better understand program behavior.

The outline and contributions of this paper are as follows:

- We argue that debugging aspect-enabled programs is more difficult and possibly more important, than debugging conventionally composed programs.
- We present a general model for discussing debugging aspect-enabled programs. The model includes a classification of AOP-specific composition techniques and AOP-specific program behaviors, and a fault model. We define the properties of an ideal AOP debugging solution, including support for *debug obliviousness* and *debug intimacy*. (§2)
- We evaluate several current AOP systems as to how well they support AOP debugging. (§3)
- Since many AOP systems employ source or binary code transformations, we consider how this affects *source-level debugging*, and present solutions suggested by related research on debugging optimized code. (§4)
- We present Wicca, our dynamic AOP system that employs a novel weaving strategy to provide full source-level debugging, and is the first dynamic AOP system to do so (§5). We present the results of a debugging experiment using Wicca that demonstrates its unique AOP debugging capabilities. (§6)

2 A Debug Model for AOP

Our AOP debug model has five components: a classification of AOP-specific composition techniques (*weaving strategies*), a classification of AOP-specific program behaviors (*AOP activities*), a fault model, a definition for debug obliviousness, and a set of debugging criteria.

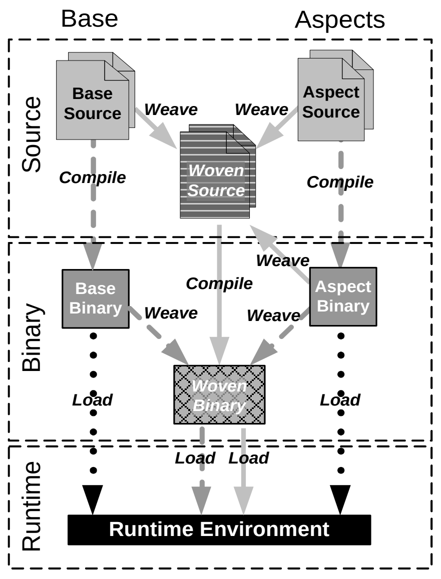
2.1 A Classification of Weaving Strategies

The AOP-specific composition technique, i.e., *weaving strategy*, used by an AOP system has a strong impact on its debuggability. Weaving is classified as either *invasive* or *noninvasive*, depending upon whether or not it performs a transformation of the base program code to enable aspect functionality. Invasive systems are further classified into *source weavers* and *binary (byte-code or machine-code) weavers*. Noninvasive systems are classified by whether they use a *custom runtime environment* or *interception*. Figure 1 depicts how the different dimensions of the weaving strategies are related.

During *source weaving* (the solid line in Figure 1), aspects are woven into the program by performing a source-to-source transformation, usually by transforming the abstract syntax tree representation of the program. The woven source is then compiled to create the final program. Because the aspect code is woven directly into the source code, it is possible to perform full source-level debugging on the aspect code using standard debuggers.

A downside of *binary weaving* (the dashed line in Figure 1) is that debug information may be invalidated by the weaving process or unavailable for injected code [2, 25]. Furthermore, companies like Microsoft have based their technical support on the assumption that an executable file and its associated attributes (date, size, checksum, and version) are fixed. Invasive weaving breaks that assumption.

Extensions to the runtime environment (the dotted line in Figure 1), e.g., AOP-enabled virtual machines and call interception plug-ins, enable aspect functionality noninvasively, i.e., without modifying the base program. Unfortunately, aspect-related behavior that is implemented in the extension may be difficult to debug.



Path	Weaving Model	Examples
➔	Source-level weaving	AspectJ, Wicca v1
– ➔	Binary weaving	AspectJ, AspectWerkz, Wicca v1
• • ➔	Custom runtime or Interception	Steamloom

Fig. 1. The relationships between different AOP weaving strategies

2.2 A Classification of AOP Activities

An *AOP activity* is any program behavior that occurs either inside the base program or inside some AOP infrastructure in support of a concept from the AOP semantic model. We use the AspectJ semantic model [27] as our reference. Table 1 categorizes the AOP activities that we have gathered from studying a wide-variety of AOP systems. Some activities, such as advice execution, map naturally to AspectJ-like language semantics, while others are common implementation approaches for supporting those semantics. Different AOP systems may combine or omit some activities. For the purposes of this paper, to qualify as an AOP system the only required activity is *advice execution*, which corresponds with the definition in [13].

We do not attempt to classify all AOP-related behavior. The level of granularity chosen is designed to be widely applicable while at the same time able to differentiate

Table 1. AOP activities that programmers would like to be able to debug

Activity	**Purpose**	**Examples**
Dynamic aspect selection	Determines at runtime which aspects apply and when.	Dynamic residue (if, instanceof, and cflow residue left over by dynamic cross-cuts) [4, 21]. Can involve runtime reflection or calls into the AOP system. Includes join point context reification [18].
Aspect instantiation	Instantiates or selects aspect instances to fulfill deployment/scoping semantics [21].	"Per" deployment semantics [21], instance-level advising, and aspect factories.
Aspect activation	Alters control flow to execute advice and provides access to join point context.	Advice method call, inlined advice code, runtime interception [31], dynamic proxies [7], and trampolines [29].
Advice execution	Execution of the advice body.	Inlined code, method call
Bookkeeping	Maintains additional AOP dynamic state.	Thread-local stack for cflow pointcuts [21].
Static scaffolding	Static modifications to the program's code, type system, or metadata.	Introductions needed to support intertype declarations, per-clause aspects, mixins, and closures. Code hoisting. [7, 21]

AOP systems based on their varied debug capabilities. The terminology is general enough to apply to other advanced techniques for the separation of concerns, including multi-dimensional separation of concerns (Hyper/J), composition filters, adaptive programming, and subject-oriented programming.

2.3 Fault Model

Each of the AOP activities in Table 1 introduces the possibility for new types of faults that were absent from the base program. Alexander et al. [2] specified a fault model for AOP that classified the new types of faults that AOP introduces that are distinct from the fault models of object-oriented and procedural programming languages. These AOP fault types were later extended by Ceccato et al. [8]. We build upon their work by generalizing and consolidating some of these fault types, by adding two of our own (*object identity errors* and *incorrect join point context*), and by associating the fault types with the AOP activities that may exhibit them.

Incorrect pointcut descriptor or advice declaration – A pointcut does not match a join point when expected, or the advice type (e.g., before, around), pointcut type (e.g., call, execution) or deployment type (e.g., "per" semantics) are incorrect. *Exhibited by activities*: dynamic aspect selection, aspect instantiation, and aspect activation.

Incorrect aspect composition – Multiple aspects that match the same join point are executed in the wrong order. *Exhibited by activities*: dynamic aspect selection, aspect instantiation, and aspect activation.

Failure to establish expected postconditions or preserve state invariants – Advice behavior or AOP activity causes a postcondition or state invariant of the base program to be violated. *Exhibited by activities*: advice execution. However, this fault can be caused by a faulty implementation of any AOP activity.

Incorrect focus of control flow – A pointcut that depends on dynamic context information, e.g., the call stack, does not match a join point when expected. The *cflow* and *if* pointcut types are examples. *Exhibited by activities*: dynamic aspect selection, aspect activation, and bookkeeping.

Incorrect changes in control dependencies – Advice changes the control flow in a way that causes the base program to malfunction. For example, adding a method override changes the dynamic target of a virtual method call. *Exhibited by activities*: aspect activation, advice execution, and static scaffolding.

Incorrect changes in exceptional control flow – Exceptions that are thrown or handled differently than they were in the base program may cause new unhandled exceptions to be thrown or prevent the original exception handlers from being called. *Exhibited by activities*: dynamic aspect selection, aspect activation, and bookkeeping.

Object identity errors – Type modifications (intertype declarations) or proxies break functionality related to object identity such as reflection, serialization, persistence, object equality, runtime type identification, self-calls, etc. *Exhibited by activities*: static scaffolding.

Incorrect join point context – The join point context available to a piece of advice is incorrect due to faulty context binding or reification. *Exhibited by activities*: dynamic aspect selection, aspect activation, and advice execution.

This list can be extended to include more fault types. The main idea is that AOP activity can introduce new types of faults that need to be debugged. We measure the debuggability of an AOP system by how easy it is to diagnose these faults. However, we will see in the next section that debuggability is at odds with the programmer's desire to remain oblivious of AOP activities.

2.4 Debug Obliviousness and Intimacy

When debugging an aspect-enabled program, the goal of *debug obliviousness* is to maintain a view of the program *as if no weaving has taken place*. Obliviousness is the primary goal for debugging optimized programs [20] as well as programs that use software dynamic translation [29] because these transformations preserve the semantics of the original program. Despite the relative importance attached to this goal [15], we are aware of no AOP system that fully supports obliviousness during debugging. The only alternative is to debug the original (non-aspect-enabled) program. However, the original program may not be available, or may require some aspects to function correctly.

Debug obliviousness is difficult to attain for invasive AOP systems because the debugger cannot distinguish between (untangle) the aspect and base program code [19]. Noninvasive systems, on the other hand, hide most aspect-related behavior by default. They still need to inform the debugging process, however, so that control flow changes related to aspect execution are also hidden. Otherwise, stepping through source code in the debugger results in unexpected jumps into aspect code. Complete obliviousness will not be possible in cases where the program's join points are entirely bypassed, for example, when *around* advice does not invoke the original join point.

Debug obliviousness becomes a liability when trying to diagnose a fault introduced by the AOP system. In this situation, we desire *debug intimacy*, the converse of debug obliviousness.

2.5 Properties of an Ideal Debugging Solution

An ideal AOP debugging solution will support debugging of all AOP activity when required or desired, and complete obliviousness otherwise. The properties of an ideal debugging solution for AOP are

(P1) ***Idempotence*** – Preservation of the base program's debug information. Idempotence ensures that whatever debug information was available before aspects were added to the base program is also available after. Noninvasive systems do not modify the original program at all. AspectJ and our Wicca system are examples of invasive systems that use source and binary weaving and ensure the debug information is maintained.

(P2) ***Debug obliviousness*** – The ability to hide AOP activity during debugging so programmers only see the base program's behavior and code.

(P3) ***Debug intimacy*** – The ability to debug all AOP activity including injected and synthesized code.

(P4) ***Dynamism*** – The ability to enable/disable aspects at runtime. When a fault occurs, the process of elimination can be used to rule out specific aspects.

(P5) ***Aspect introduction*** – The ability to introduce new aspects, e.g., debugging and testing aspects, in an unanticipated fashion. An example of this is *dynamic aspect introduction* that allows aspects to be introduced without restarting.

(P6) ***Runtime modification*** (also called *edit-and-continue*) – The ability to modify base or aspect code at runtime, e.g., to quickly add a *printf* statement, enable tracing, or try out a bug fix, without restarting. This is useful for interactive debugging and for diagnosing hard-to-reproduce bugs.

(P7) ***Fault isolation*** – The ability for the debugger to automatically determine if a fault lies within the base code, advice code, or some other AOP activity code. Invasive weavers may invalidate the traditional assumption that library boundaries establish ownership since AOP-related code or metadata, possibly written by a third party, is intermingled with the base program [19].

3. An Evaluation of the Debuggability of Existing AOP Systems

In Table 2, we show the results of our evaluation of a representative sample of AOP systems based on our ideal debugging properties.

Static AOP. All the Java byte-code weavers satisfy the idempotence property, because they maintain the debug information of the original program when weaving. Java stores debug information inside the class file, alongside the class definition and

AOP Tools & Systems	Idempotence	Debug intimacy	Debug obliviousness	Dynamism	Aspect introduction	Runtime modification	Fault isolat./repud.
AOP.NET	✓	O		✓			
AOP-Engine		O		✓	✓		
Arachne	✓			✓	✓		
AspectJ	✓	O					
AspectWerkz	✓	O	¤	✓			
Axon	✓	O	¤	✓			
CaesarJ	✓						
CAMEO	✓	✓					
CLAW		O		✓			
EAOP	✓	O	¤	✓	✓		
Handi-Wrap	✓			✓	✓		
Hyper/J	✓	✓					
JAsCo		O	¤	✓	✓		
nitrO		O		✓	✓		
Wicca v1	✓	✓		✓	✓	✓	
PROSE v2	✓	O	¤	✓	✓		
SourceWeave.NET	✓	✓					
Steamloom	✓	O	¤	✓	✓		
Wool	✓	O	¤	✓	✓		

✓- Fully supported
O- Partial *advice execution* debugging supported
¤- Partial obliviousness supported

Table 2. AOP debuggability comparison matrix. Our system, Wicca, is shown in bold

byte code. The debug information is co-located with the class file, and its format is well documented, improving the likelihood that byte-code rewriters will propagate it correctly.

For Windows executables, debug information is stored in a separate program database (PDB) file that becomes invalid when the executable is transformed. Ideally, the transformation process would update the debug information but this is a very complex process. Our Wicca system is the only .NET byte-code weaver (that we are aware of) that updates the debug information, which is made possible by the Microsoft Phoenix backend compiler framework[1].

Dynamic AOP. *Invasive* dynamic AOP systems transform the base program by using dynamic proxies [7] or by injecting join point stubs (also called hooks or trampolines) at all potential join points [6, 9, 16]. These systems typically support debugging of

[1] http://research.microsoft.com/phoenix

advice execution. Aspect selection, instantiation, or activation logic, however, may be implemented inside the *dynamic AOP infrastructure* [19] and may be difficult to debug. This difficulty makes it hard to understand the woven program's control flow and diagnose problems related to aspect ordering and selection ("Why didn't my aspect run?") [2]. In addition, hook injection may invalidate the base program's debug information (violating the idempotence property), which will result in a confusing or misleading debugging experience.

Noninvasive dynamic AOP systems use a custom runtime environment (e.g., JRockit[2], Steamloom [19], PROSE [31]) or take advantage of interception services (e.g., .NET Profiler API [16], Java debugger APIs [3, 31]), to provide AOP functionality without transforming the base program. These systems have the benefit that the base program's debug information is left intact (idempotence). They suffer from the drawback that any AOP activities that are implemented as part of the runtime or native library are not debuggable. Aspect-enabled programs can be confusing to debug at the source level because control flow appears to change mysteriously; e.g., stepping into a function in the debugger results in a different function being entered. In addition, use of the Java debugger APIs to implement dynamic AOP currently prevents the application from being debugged inside a standard debugger.

4 Source-Level Debugging

Source-level debuggers strive to maintain the illusion of a source-level view of program execution. They commonly allow the programmer to set location and data breakpoints, step through code, inspect the stack, inspect and modify variables and memory, and even change the running code. To enable this, the debugger requires a correspondence between the program's compiled code and source code. This *debug information* is generated during compilation and consists of file names, instruction-to-line number mappings, and the names and memory locations of symbols. The information is usually stored inside the program executable, library, or class file, or in a separate *debug information file*. It may be absent if the build process excluded it, to lower the memory footprint for example, or if it was stripped out for the purposes of compression or obfuscation.

When compilation involves a straightforward syntax-directed translation [1], the compiler can provide a one-to-one correspondence from byte code (or machine code) and memory locations to source. The correspondence becomes more complicated as transformations are applied at various stages of the pre-processing, compilation, linking, loading, just-in-time compilation, and runtime pipeline. This lack of correspondence between the source and compiled code makes it difficult for the debugger to match the *actual behavior* of the executing code with the *expected behavior* from the source-code perspective [34], and leads to the *code location* and *data-value problems* that have been studied extensively in the context of debugging optimized code [14, 20,

[2] http://dev2dev.bea.com/jrockit

29, 34]. In the context of debugging aspect-enabled programs these problems have been mentioned but briefly [2, 7, 24, 25].

In the AOP context, we define *full source-level debugging* as the ability to perform source-level debugging on all the AOP activities listed in Table 1.

4.1 The Code Location Problem

The *code location problem* arises when transformations are applied that prevent a one-to-one correspondence between compiled code and source code. In the domain of optimizing compilers [1], the problem is caused by the removal, merging, duplication (in-lining), reordering, or interleaving of instructions. In the domain of AOP weaving, the code location problem is usually caused by the removal (e.g., hoisting [4]), insertion (e.g., code synthesis, dynamic residue, aspect method calls, aspect in-lining, closures), duplication (e.g., initialization in-lining), or reordering (e.g., due to around-advice) of instructions [21]. The problem causes the debugger to show the wrong source line or call stack, or show byte code (or machine code) instead of source code.

4.2 The Data-Value Problem

The *data-value problem* occurs when transformations obscure the correspondence between variables in the source code and locations in memory [20]. Optimizing compilers commonly fold constants, eliminate common subexpressions, and represent variables in registers instead of memory (sometimes the same storage location will represent different variables at different times). In the context of AOP, weavers may add fields to classes (*introduction*) and formal arguments and local variables to methods (e.g., for context exposure) [21]. This problem causes the debugger to show new variables or fields incorrectly, e.g., it may be missing or have the wrong name.

4.3 Possible Approaches for Supporting Source-Level Debugging

Below we have consolidated and generalized some common approaches to the problem of performing source-level debugging of aspect-enabled programs.

Source weaving [33] – Wicca, AspectJ, and SourceWeave.NET [25] are example AOP systems that use source weaving and support full source-level debugging.

Debugger-friendly weaving – Wicca, AspectJ, and AspectWerkz [7] are example AOP systems that use binary-level weaving but are able to preserve the original debug information, thus supporting the idempotence property (P1).

Annotation [5] – Refers to the ability to annotate aspect code to provide rich debug information, to allow the debugger to hide the code in support of debug obliviousness, and to support fault isolation. Although AspectJ and Steamloom [19] use byte code annotation, no AOP system that we are aware of currently uses annotation for debugging purposes.

Reverse engineering [2, 23] – When the debugger encounters byte or machine code that has no matching source information, it can hide the code if debug obliviousness is desired or synthesize the source code on-the-fly if debug intimacy is desired.

Static analysis [20] – Static analysis techniques can be used to detect injected aspect code, for example, and, similar to annotation, used to provide debug information or to support obliviousness.

To allow the programmer to be truly oblivious of the aspects composed with the program, source-level debugging must hide all AOP-related code and behavior. However, we are aware of no AOP system that fully supports this. In §6, we show how intimate source-level debugging is useful for debugging AOP-specific faults. This is akin to directing a C compiler to display preprocessed source files to diagnose problems with include files and macros. Furthermore, when the transformation technology is immature, as is the case for AOP, a source-level representation of the transformation helps implementers detect faults [29, 34].

Noninvasive AOP implementations may not weave code at all. For these implementations, the ability to debug AOP-related code at the source level is nonsensical. However, these systems can still provide support for debug obliviousness and intimacy. For example, intimacy can be supported by showing a runtime visualization of the base program and aspect behavior [17]. For obliviousness, only the base program behavior is shown.

5 Wicca

Most dynamic AOP solutions involve binary weaving, a custom runtime, dynamic proxies, or method call interception. To support full source-level debugging, Wicca takes a new approach—it performs *dynamic source weaving*.

5.1 Overview

Wicca[3] v1 is a prototype dynamic AOP system for C# applications that performs source weaving (the solid line in Figure 1) at runtime. The woven source code is compiled in the background and the running executable is patched on-the-fly [12]. The entire weave-compile-update process takes less than 2.5 seconds for a C# program with 14,531 source lines on a Pentium IV 3.6 GHz processor. Wicca v1 uses the .NET Profiler API to enable dynamic weaving and patching, which imposes a 5-7% runtime overhead on application performance when compared to running the program without aspects enabled. Wicca also supports static byte-code weaving. A more detailed description of Wicca including performance measurements can be found in the expanded version of this paper [11].

Because all AOP activities are represented in source code, the programmer can perform full source-level debugging on the woven program using *wdbg*, our custom debugger. In addition, several ancillary debugging activities are supported:

- Full source-level debugging (*idempotence* and *debug intimacy*)
- Aspects can be enabled/disabled at runtime (*dynamism*)
- Aspect rules, located in an XML file, can be changed at runtime (*dynamism*)

[3] Derived from the Old Norse word *vikja* meaning to turn, bend and shape.

- New aspects can be introduced at runtime (*aspect introduction*)
- Advice code can be modified at runtime (*runtime modification*)
- Base code can be modified at runtime (*runtime modification*)

To our knowledge, Wicca is the first dynamic AOP system to support full source-level debugging and modification of advice and base code at runtime. Although Wicca uses a radical approach, i.e., dynamic source weaving, this approach offers unique interactive source-level debugging capabilities. If the interactive capabilities are not needed, *static source weaving* [25] is a simple and sufficient alternative.

5.2 The Wicca Debugger (wdbg)

Wdbg is the first debugger we are aware of that supports source-level debugging of dynamically updated programs. It is an extension to the Microsoft cordbg command-line debugger. An extension was required because standard Windows debuggers do not support dynamically changing the debug information associated with the application being debugged. Without this extension, the source code and variables displayed in the debugger may be incorrect. Static weavers do not have to deal with this issue.

5.3 Limitations

Wicca v1 has limited AOP functionality. Only before and after advice, and method execution and field access join points, are supported. Introductions (inter-type declarations) are not supported. Wicca v1 also requires source code for both the base program and the aspects. While Wicca v1 does not support debug obliviousness, this could be achieved using our statement annotation technology [10].

Due to a limitation of the Profiler API, we are not able to update a function that is active on the stack. The function is updated the next time it is called. Unfortunately, wdbg will incorrectly show the woven source code instead of the original source code, if the function has been updated yet. We expect the fix for this to be straightforward.

6 Evaluation

In this section we present the results of an experiment to demonstrate the interactive debugging capabilities of Wicca.

6.1 Experimental Setup

We are given a buggy C# class that is supposed to implement a stack (see Listing 1) and a test driver for exercising the stack class. We will use Wicca to interactively diagnose and fix the bugs. To help diagnose the bug, we create an aspect that embodies the *design-by-contract* (DBC) [30] principle. DBC allows the programmer to make assertions [22] about the system, in the form of *preconditions*, *postconditions*, and *class invariants*. For example, the class invariant for the stack class is that the top element of a non-empty stack must not be null. Its push method has a precondition that the object being pushed is non-null, and a postcondition that the stack's size has been incremented.

Normally, the assertion checking and handling code is scattered throughout the system. By localizing the assertion code into a DBC aspect (Listing 2), we obtain many benefits including improved code clarity, the ability to easily change the assertion violation policy, to strengthen or weaken class invariants, to add assertions to a class after-the-fact, and to automate contract enforcement. [26] Moreover, unlike normal assertions which are only checked for debug builds, or which require continuous checking at runtime, Wicca can inject these *test probes* [22] on demand, thus completely eliminating checking overhead when assertions are disabled.

```
public class Stack {
   ArrayList elements = new ArrayList();
   public void push(object arg1) {
      elements.Add(arg1);
      elements.Add(arg1); // <-- Bug!
   }
   public object pop() {
      object popped = top();
      elements.RemoveAt(elements.Count-1);
      return popped;
   }
   public object top() {
      return elements[elements.Count-1];
   }
   ...
```

Listing 1. A stack class written in C# that contains a bug in the push() method

6.2 Detecting Faults using Test Probes

To test the stack class we create a test driver that pushes several items onto the stack and then pops each one while writing its value to the console. Shortly after launching the test driver, we notice a bug (see Listing 1) where every item in the stack is duplicated. While the driver is running, we enable the stack DBC aspect, which may already exist or which we may have introduced for this debugging task. Wicca detects this change and *rebuilds* (reparses, reweaves, and recompiles) the driver, taking a total of 610 ms on a Pentium IV 3.6 GHz processor.

Listing 3 shows the aspect rule file after we added the stack DBC aspect and enabled weaving. Immediately, the aspect code detects a postcondition violation and

```
public class StackDBCAspect {
  static int __savedCount;

  static void PostCond_push(Stack __this, object arg1) {
    if (__this.isEmpty())
      throw new InvalidOperationException(
        "Postcondition violated: Stack is empty after push");
    if (__this.top() != arg1)
      throw new InvalidOperationException(
        "Postcondition violated: Pushed item is not on top of stack");
    if (__this.count() != __savedCount + 1)
      throw new InvalidOperationException(
        "Postcondition violated: Stack size did not increase " +
        "by one after push");
   }
    ...pre and postconditions for pop, etc...
```

Listing 2. A design-by-contract aspect for the stack class. Variables that start with "__" are renamed during weaving

```
<aspects enable="true">
  <aspect class="StackDBCAspect" sources="StackDBCAspect.cs">
    <advice name="PreCond_push" type="after">
      <pointcut expression="execution(Stack.push())" />
    </advice>
    <advice name="PostCond_push" type="before">
      <pointcut expression="execution(Stack.push())" />
    </advice>
```

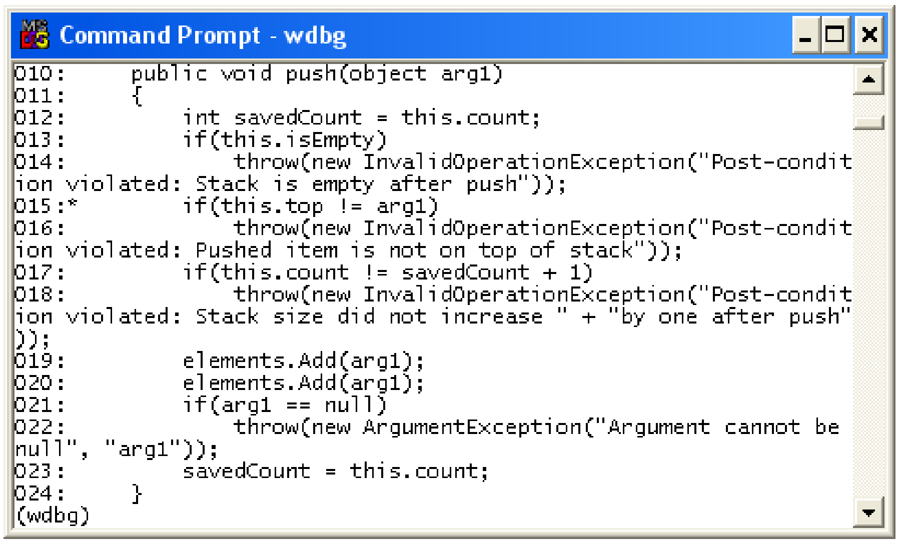
Listing 3. Aspect rule file with erroneous before and after advice.

throws the exception: “Postcondition violated: Stack is empty after push.” The exception message provides the file name and line number where the exception occurred.

6.3 Just-In-Time Debugging

We launch the Wicca debugger, wdbg, to debug the exception. After pointing wdbg to the debug information of the woven program, we can step into the push method and see the interwoven source code (see Figure 2). What is significant about this figure is that the base program and all AOP activities are debuggable at the source level.

Looking at the source code for the push method, it is obvious that there are actually *two* bugs: the precondition and postcondition are switched and the Add method is called twice. The first bug is a manifestation of an AOP-specific fault: *incorrect pointcut descriptor*. This fault is difficult to diagnose without a source-level representation of the woven code. From the woven code it appears that the postcondition and precondition are switched. Looking closely at the aspect rules in Listing 3 reveals that the push precondition (PreCond_push) is erroneous because the advice type is “after” when it should actually be “before”, and similarly for the postcondition.



```
010:     public void push(object arg1)
011:     {
012:         int savedCount = this.count;
013:         if(this.isEmpty)
014:             throw(new InvalidOperationException("Post-condit
ion violated: Stack is empty after push"));
015:*        if(this.top != arg1)
016:             throw(new InvalidOperationException("Post-condit
ion violated: Pushed item is not on top of stack"));
017:         if(this.count != savedCount + 1)
018:             throw(new InvalidOperationException("Post-condit
ion violated: Stack size did not increase " + "by one after push"
));
019:         elements.Add(arg1);
020:         elements.Add(arg1);
021:         if(arg1 == null)
022:             throw(new ArgumentException("Argument cannot be
null", "arg1"));
023:         savedCount = this.count;
024:     }
(wdbg)
```

Fig. 2. A wdbg debugging session showing aspect code interwoven with the stack class. The asterisk (*) indicates the current line.

A quick change to the aspect rules to fix this oversight causes Wicca to reparse, re-weave, and recompile the driver. As expected, an exception is thrown immediately but this time with the correct message: "Postcondition violated: Stack size did not increase by one after push." After removing the extraneous Add method call, Wicca rebuilds the driver, and we immediately see the correct behavior. *At no time during the debugging session did we have to restart the test driver.*

7 Related Work

A few systems deserve further comment. SourceWeave.NET [25] employs a very similar source weaving strategy that is designed to improve source-level debugging. However, it weaves statically whereas Wicca weaves dynamically, enabling aspects to be introduced and reconfigured at runtime.

Few AOP systems support debug obliviousness or fault isolation, which requires a debugger to identify AOP activity code. AspectJ and Steamloom support byte-code annotations for identifying aspects to prevent recursion during weaving [21] and to facilitate aspect removal [19]. As far as we know, no AOP system uses byte-code annotations to support obliviousness or fault isolation.

8 Conclusion

We described the problem of debugging aspect-enabled programs and why it has become an important gating criterion for the adoption of AOP. We provided a debug model for AOP that classified all AOP activities, related them to the new type of faults they can introduce, outlined the properties of an ideal debugging solution, and surveyed the state of the art of AOP debugging. For source-level debugging, we explained how the nature of binary weavers gives rise to the *code location problem*, that originates from the field of optimizing compilers. We showed how results from that community apply to debugging aspect-enabled programs.

We demonstrated how our Wicca system offers a novel approach to debugging dynamically composed aspect-enabled programs. Wicca is the first dynamic AOP system to support full source-level debugging. It does this by employing a novel *dynamic source weaving* strategy that combines source weaving with online byte-code patching with relatively low overhead. Our future work will be to explore using byte-code annotations [10] to fully support debug obliviousness and fault isolation.

Acknowledgements

We thank Mike Stall, John Lefor, Chuck Mitchell, Jan Gray, Sonja Keserovic, and Andy Ayers from Microsoft for their support and feedback. We also thank Gregor Kiczales, Hrvoje Benko, Rean Griffith, and our anonymous reviewers for their feedback. This research is funded in part by a grant from Microsoft Research.

References

[1] A. Aho, M. Lam, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools, second edition*. Addison-Wesley, 2007.

[2] R. Alexander, J.M. Bieman, and A. Andrews. Towards the Systematic Testing of Aspect-Oriented Programs. Tech Rep CS-4-105. Dept. of CS, Colorado State Univ., Mar. 2004.

[3] S. Aussmann and M. Haupt. Axon – Dynamic AOP through Runtime Inspection and Monitoring. In Proc. of the Wkshp. on Advancing the State-of-the-Art in Runtime Inspection (ASARTI'03), July 2003.

[4] P. Avgustinov, A.S. Christensen, L. Hendren, S. Kuzins, J. Lhoták, O. Lhoták, O. de Moor, D. Sereni, G. Sittampalam, and J. Tibble. Optimising AspectJ. In Proc. of Prog. Language Design and Implementation (PLDI'05), June 2005.

[5] J. Van Baalen, P. Robinson, M. Lowry, T. Pressburger. Explaining Synthesized Software. In Proc. of Automated Software Eng. (ASE'98), Oct. 1998.

[6] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In Proc. of Aspect-Oriented Software Development (AOSD'02), Apr. 2002.

[7] J. Bonér. AspectWerkz — dynamic AOP for Java. Invited talk at Aspect-Oriented Software Development (AOSD'04), Mar. 2004.

[8] M. Ceccato, P. Tonella and F. Ricca. Is AOP code easier or harder to test than OOP code? In Proc. of the Wkshp. on Testing Aspect-Oriented Programs (WTAOP'05), Mar. 2005.

[9] R. Douence, T. Fritz, N. Loriant, J.-M. Menaud, M. Segura-Devillechaise and M. Südholt. An expressive aspect language for system applications with Arachne. In Proc. of Aspect-Oriented Software Development (AOSD'05), Mar. 2005.

[10] M. Eaddy and A. Aho. Statement Annotations for Fine-Grained Advising. In Proc. of the Wkshp. on Reflection, AOP, and Meta-data for Software Evol. (RAM-SE'06), July 2006.

[11] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging Woven Code. Tech Rep. CUCS-035-06. Dept. of CS, Columbia Univ., Sept. 2006.

[12] M. Eaddy and S. Feiner. Multi-Language Edit-and-Continue for the Masses. Tech Rep CUCS-015-05. Dept. of CS, Columbia Univ., Apr. 2005.

[13] T. Elrad, R. Filman, and A. Bader, *Aspect-oriented programming: Introduction*. Communications of the ACM. 44(10): p. 29-32, 2001.

[14] R. Faith. Debugging Programs after Structure-Changing Transformation. Ph.D. dissertation, CS Dept., Univ. of North Carolina, Dec. 1997.

[15] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In OOPSLA Wkshp. on Advanced Separation of Concerns, Oct. 2000.

[16] A. Frei, P. Grawehr, and G. Alonso. A Dynamic AOP-Engine for .NET. Tech Rep 445. Dept. of CS, ETH Zürich, Mar. 2004.

[17] W. G. Griswold, J. Yuan, and Y. Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. In Proc. of the Intl. Conf. on Software Eng. (ICSE '01), May 2001.

[18] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. Proc. of Net.ObjectDays. Springer-Verlag LNCS 3263, pp. 81-96, Sept. 2004.

[19] M. Haupt, M. Mezini, C. Bockisch, T. Dinkelaker, M. Eichberg and M. Krebs. An Execution Layer for Aspect-Oriented Prog. Languages. In Proc. of Virtual Execution Environments (VEE'05), June 2005.

[20] J. Hennessy. *Symbolic Debugging of Optimized Code*. ACM Transactions on Prog. Languages and Systems, 4(3): 323-344, July 1982.

[21] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In Proc. of Aspect-Oriented Software Development (AOSD'04), Mar. 2004.

[22] C. A. R. Hoare. *Assertions: a personal perspective. Software pioneers: contributions to software engineering*, Springer-Verlag, pp. 356-366, 2002.

[23] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. Proc. of Prog. Language Design and Implementation (PLDI'05), July 1992.

[24] J. Hugunin. The next steps for aspect-oriented programming languages (in Java). In Proc. of Wkshp. on New Visions for Software Design & Prod.: Research & Apps., Dec. 2001.

[25] A. Jackson and S. Clarke. SourceWeave.NET: Source-level cross-language aspect-oriented programming. In Proc. of Generative Prog. and Component Eng. (GPCE'04), Oct. 2004.

[26] M. Lippert and C. V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In Proc. of the Intl. Conf. Software Eng. (ICSE'00), June 2000.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In Proc. of the European Conf. on Object-Oriented Prog. (ECOOP'01), Springer-Verlag LNCS 2072, pp. 327-353, Berlin, Germany, June 2001.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. TR SPL97-008 P9710042, Xerox PARC, Feb. 1997.

[29] N. Kumar, B. Childers and M. L. Soffa. Tdb: a source-level debugger for dynamically translated programs. In Proc. of the Intl. Symp. on Automated and Analysis-Driven Debugging (AADEBUG'05), Sept. 2005.

[30] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New Jersey, 1997.

[31] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. In Proc. of the Wkshp. on Adaptive and Self-Managing Enterprise Applications (ASMEA'05), June 2005.

[32] S. Redwine and W. Riddle. Software technology maturation. In Proc. of Software Eng. (SE'85), Aug. 1985.

[33] C. Tice and S. Graham. OPTVIEW: A New Approach for Examining Optimized Code. Wkshp. on Program Analysis for Software Tools and Eng. (PASTE'98), June 1998.

[34] P. T. Zellweger. Interactive Source-Level Debugging of Optimized Programs. Ph.D. dissertation, CS Dept., Univ. of California, Berkeley. Also published as Xerox PARC Tech. Rep. CSL-84-5, May 1984.