

Debugging Woven Code

Marc Eaddy¹, Alfred Aho¹, Weiping Hu², Paddy McDonald², Julian Burger²

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

September 20, 2006

Abstract. The ability to debug woven programs is critical to the adoption of Aspect Oriented Programming (AOP). Nevertheless, many AOP systems lack adequate support for debugging, making it difficult to diagnose faults and understand the program's structure and control flow. We discuss why debugging aspect behavior is hard and how harvesting results from related research on debugging optimized code can make the problem more tractable. We also specify general debugging criteria that we feel all AOP systems should support.

We present a novel solution to the problem of debugging aspect-enabled programs. Our Wicca system is the first dynamic AOP system to support full source-level debugging of woven code. It introduces a new weaving strategy that combines source weaving with online byte-code patching. Changes to the aspect rules, or base or aspect source code are rewoven and recompiled on-the-fly. We present the results of an experiment that show how these features provide the programmer with a powerful interactive debugging experience with relatively little overhead.

1 Introduction

We use the term *debuggability*¹ to mean the ability to diagnose faults in a software system, and to improve comprehension of a system, *by monitoring the execution of the*

¹ The term *debugging expressiveness* is used in [28].

system. Various debugging techniques are possible including source-level debugging, *printf*-style debugging, assertions, tracing, logging, and runtime visualization.

Debugging 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.

AOP is still an emerging field with many different techniques for aspect specification, weaving, composition, and integration. Along with tool support, debugging support serves as an indicator of AOP maturity [20, 38]. 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, 20, 27, 28].

Debugging is no substitute for *aspect visualization* [20] 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 contributions of this paper are as follows:

- We identify the *AOP debugging problem* and explain why it is important and difficult to solve. (§3)
- We define *AOP debuggability* as a classification of AOP activities and the AOP fault types they induce, and the properties of an ideal AOP debugging solution, including support for *debug obliviousness*. 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 focus on 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)
- Wicca includes a static byte-code weaver which is the first to preserve debug information when weaving .NET executables. We explain why this is significantly harder to accomplish on the .NET platform than on the Java™ platform. (§5)
- We suggest general strategies for debugging synthesized code and for supporting debug obliviousness. (§8)

In the next section, we define aspect-oriented programming and lay the groundwork for evaluating the debug capability of an AOP system.

2 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) [31] can improve the *separation of concerns* in software systems when traditional techniques fail. Software requirements that require code to be scattered across the entire system and intermingled (tangled) with otherwise unrelated code are considered *crosscutting concerns*. Nonfunctional (operational) requirements, such as tracing, profiling, transactions, auditing, persistence, error handling, and quality-of-service, are often crosscutting in nature.

AOP introduces the concept of an *aspect* that modularizes all the code necessary to implement a crosscutting concern, thus untangling the system and making it easier to develop, understand, and maintain. The aspect functionality is enabled by *weaving* the aspect code into the system in an automated way based on programmer-specified *aspect rules*.

2.1 Semantic Model

We use the language model from AspectJ™ [30], the most prominent AOP language, to define the semantic model we use when explaining the debuggability of woven programs. AOP allows aspect functionality to be executed at specific points in the dynamic call graph of a running program, called *join points* [24]. Example join points are method call/execution, constructor call/execution, field read/write, object initialization, and thrown/handled exception. An aspect consists of a *pointcut* that identifies specific join points, the code to be executed at those join points (*advice*), and whether the advice will be executed before (*before advice*), after (*after advice*), or instead of (*around advice*) the join point. The *aspect deployment model* specifies which advice instances map to which join points. For example, aspect instances can be per-call, per-method, per-object, per-thread, per-class, or singleton.

2.2 Weaving Strategies

The 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.

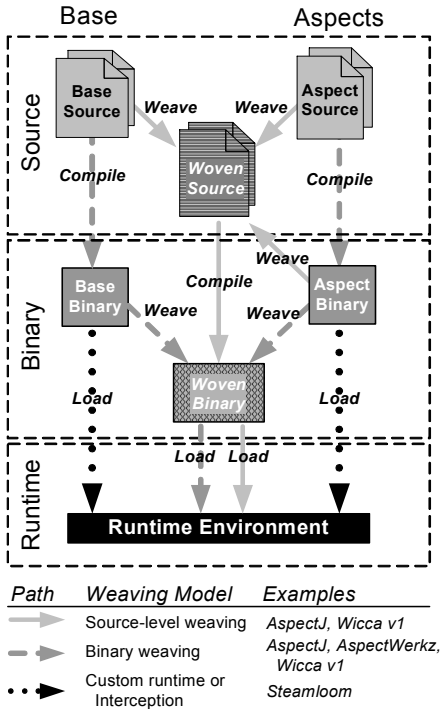


Fig. 1. The relationships between different AOP weaving strategies

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.

Some benefits of binary weaving (the dashed line in Figure 1) are that programs can be woven even if source is unavailable (or when compilation is not desired); weaving can be delayed until load-time, JIT-time, or runtime; and multiple languages may be supported. A downside is that debug information may be invalidated by the weaving process or unavailable for injected code [2, 28]. Furthermore, companies like Microsoft have based their support infrastructure around the assumption that an executable file and its associated attributes (date, size, checksum, and version) are not changed. 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

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

AOP 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 crosscuts [7]) [4, 24]. Can involve runtime reflection or calls into the AOP system. Includes join point context reification [21].
Aspect instantiation	Instantiates or selects aspect instances to fulfill aspect deployment/scoping semantics. Also known as “advice-instance lookup” [24].	Per-call, per-method, per-object, per-class, and singleton deployment semantics [24], instance-level advising, and aspect factories.
Aspect activation	Alters control flow to execute advice and provides access to join point context.	Method call instruction, transition to inlined advice code, runtime interception machinery [37], dynamic proxies [35], and trampolines [34].
Advice execution	Execution of the advice body.	Inlined code, method call
Bookkeeping	Maintains additional AOP dynamic state.	Thread-local stack for cflow pointcuts [24].
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. [8, 24]

noninvasively, i.e., without modifying the base program. They can provide ubiquitous and efficient support for AOP. Aspect-related behavior that is implemented in the extension, however, may be difficult to debug.

2.3 A Classification of AOP Activities

An *AOP activity* is any code, either inside the base program or part of some AOP infrastructure, in support of some concept from the AOP semantic model. 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 AOP semantics, while others are common implementation approaches to supporting the semantics. Since this is AspectJ’s semantic model, we expect that AspectJ performs all these activities. Other 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*, corresponding with the definition in [14].

We do not attempt to classify all AOP-related code. The level of granularity chosen is designed to be widely applicable while at the same time able to differentiate AOP systems based on their varied debug capabilities.² 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.

3 A Debug Model for AOP

We present a debug model for AOP that has three components: a fault model, a definition for debug obliviousness, and a set of debugging criteria. We show how AOP activities can introduce new fault types in the base program. We also evaluate how well some current AOP systems measure up to our debugging criteria.

3.1 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 faults models of object-oriented and procedural programming languages. These AOP fault types were later extended by Ceccato et al. [9]. We build upon their work by generalizing and consolidating some of these fault types, by adding one of our own, 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 (before, after, around), pointcut type (e.g., call, execution), or deployment type (e.g., per-this, per-thread) are incorrect. *Exhibited by activities:* dynamic aspect selection, aspect instantiation, and aspect activation.

Incorrect aspect precedence – 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.

² Researchers sometimes call all AOP activity code other than advice execution *aspect glue code* or *aspect decorations* [16].

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, comparison, 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 point is that AOP activity can introduce new types of faults that will need to be debugged. We measure the debuggability of an AOP system by determining 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.

3.2 Debug Obliviousness and Intimacy

We specialize *obliviousness* as defined by Filman and Friedman [16] by introducing the concept of *debug obliviousness*, which pertains to the level of awareness and monitoring ability that the programmer has of AOP activity during debugging.

When debugging aspect-enabled code, 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 [23] as well as programs that use software dynamic translation [34] because these transformations preserve the semantics of the original program. Despite the relative importance attached to this goal, we are aware of no AOP system that fully supports obliviousness during debugging.

Debug obliviousness is difficult to attain for invasive AOP systems because the debugger cannot distinguish between (untangle) the aspect and base program code [22]. 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.³

While AOP researchers usually espouse the virtues of obliviousness in general, it becomes a liability when trying to diagnose a fault introduced by the AOP system. In this situation, we may need *debug intimacy*, the converse of debug obliviousness.

3.3 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.
- (P2) **Debug obliviousness** – The ability to hide AOP activity during debugging so the programmer only sees their code only.
- (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, for example, debugging and testing aspects, in an unanticipated fashion. An example of this is *dynamic aspect introduction*, sometimes called *aspectual polymorphism* [7] 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.

Idempotence ensures that whatever debug information was available before the base program was aspect-enabled 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.

We discussed debug obliviousness and intimacy in §3.2. Properties P4-P6 are not as critical but are nonetheless useful for debugging. We explain why in §5.

Fault isolation is supported by the Windows platform for applications that use dynamic link libraries (DLLs). If an exception occurs in a library, the exception information is displayed as well as the name of the offending DLL. The user is allowed to

³ The same phenomenon exists when debugging multi-threaded code; control flow may jump from one thread to another unexpectedly. To prevent this, the debugger may disable thread switching during debugging.

upload the program’s crash data to Microsoft for analysis and may even be able to download a patch, thus enabling immediate fault remediation.

Similar to debug obliviousness, fault isolation is complicated by invasive weaving. Invasive aspect weavers can 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 [22]. Annotating aspect code is one possible solution that we discuss in §4.3 and §8.1.

3.4 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 the ideal debugging properties. Some common themes are apparent and we discuss these next.

Static Weavers. 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 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 API⁴.

Dynamic AOP. *Invasive* dynamic AOP systems transform the base program by using dynamic proxies [8, 35] or by injecting join point stubs (also called hooks or trampolines) at all potential join points [6, 11, 12, 19]. These systems typically support debugging of advice execution. Aspect selection, instantiation, or activation logic, however, may be implemented inside the *dynamic AOP infrastructure* [22] 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.

⁴ <http://research.microsoft.com/phoenix>

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

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

- ✓ - Fully supported
- - Partial *advice execution* debugging supported
- ⊠ - Partial obliviousness supported

Noninvasive dynamic AOP systems use a custom runtime environment (e.g., JRocket⁵, Steamloom [22], PROSE [37]) or take advantage of interception services (e.g., .NET Profiler API [19, 25], Java debugger APIs [3, 37]), 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

⁵ <http://dev2dev.bea.com/jrocket>

addition, use of the Java debugger APIs to implement dynamic AOP currently prevents the application from being debugged inside a standard debugger.

In the next section, we position source-level debugging as one technique for debugging AOP activity, and outline its challenges and some possible solutions.

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 relationship becomes more complicated as transformations are applied at various stages in 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 [41], and leads to the *code location* and *data-value problems* that have been studied extensively in the context of debugging optimized code [15, 23, 34, 40, 41]. In the context of debugging woven code these problems have been mentioned but briefly [2, 7, 17, 18, 27, 28, 32].

The same issues with debugging transformed code have surfaced in several other fields including term rewriting systems where it is called the *origin tracking problem* [10], algorithm animation where it is called the *execution animation problem*, and from generative programming where it is called the *source-object correlation problem* and *transformation tracking problem* [33].

We would like to leverage this work and, where possible, reuse approaches, results, and terminology. In the next section, we generalize the code location and data-value problems to include all types of program transformation, including those performed by invasive AOP weavers.

4.1 The Code Location and Data-Value Problems

The *code location problem* arises when program 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., around-advice) of instructions [24]. 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.

The *data-value problem* occurs when program transformations obscure the correspondence between variables in the source code and locations in memory [23]. 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) [24]. This problem causes the debugger to show new variables or fields incorrectly, e.g., a field may be missing or have the wrong name.

4.2 Possible Solutions

Below we have consolidated and generalized some common approaches to the problem of performing source-level debugging of woven code

- (T1) **Disable weaving** [23]
- (T2) **Source weaving** [39] – AspectJ, Wicca and SourceWeave.NET [28] are example AOP systems that use source weaving and support full source-level debugging.
- (T3) **Debugger-friendly weaving** – Wicca, AspectJ, and AspectWerkz [8] are example AOP systems that use binary-level weaving but are able to preserve the original debug information, thus supporting the idempotence property (P1).
- (T4) **Annotation** [5, 10, 22] – Refers to the ability to annotate aspect code to provide rich debug information or to allow the debugger to hide the code in support of debug obliviousness, or for fault isolation. Although AspectJ and Steamloom [22] use byte code annotation, no AOP system that we are aware of currently uses annotation for debugging purposes.
- (T5) **Reverse engineering** [2, 26] – When the debugger encounters byte code 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.

(T6) *Static analysis* [23] – 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.

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. In the next section we describe our system, Wicca, that leverages solutions T1, T2, and T3, to support full source-level debugging.

5 Wicca

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

Wicca⁶ 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. The entire weave-compile-update process takes less than 2.5 seconds on the programs we have tested and our system imposes a modest 5-7% runtime overhead on application performance. We provide a breakdown of the performance overheads in §6.4.

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:

1. Full source-level debugging (*idempotence* and *debug intimacy*)
2. Aspects can be enabled/disabled at runtime (*dynamism*)
3. Aspect rules, located in an XML file, can be changed at runtime (*dynamism*)
4. New aspects can be introduced at runtime (*aspect introduction*)
5. Advice code can be modified at runtime (*runtime modification*)
6. Base code can be modified at runtime (*runtime modification*)

To our knowledge, full source-level debugging and modification of advice and base code at runtime are not supported by any other dynamic AOP system.

⁶ Derived from the Old Norse word *vikja* meaning to turn, bend and shape.

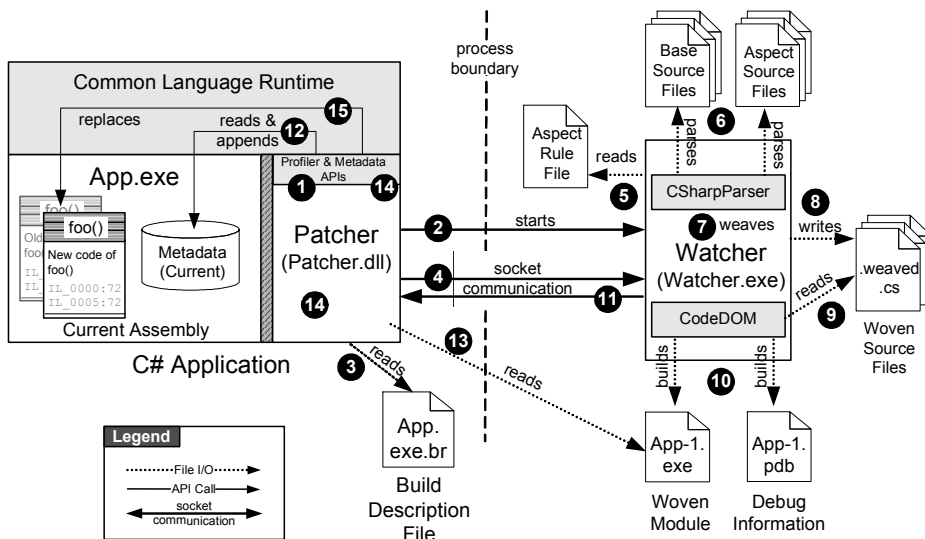


Fig. 2. Wicca v1 – Dynamic weaving walkthrough

5.1 Walkthrough

Figure 2 shows a Wicca walkthrough involving load-time weaving, with numbers indicating the interaction sequence. For more details, see [13].

Load-Time Weaving. Wicca consists of three subsystems: Patcher, Watcher, and Phx.Morph.⁷ When the application starts, the .NET runtime loads Patcher which uses the Profiler API to hook application events (1). Patcher spawns Watcher (2), and then reads the *build description file* associated with each module to obtain the source files, referenced assemblies, and compiler options (3). Patcher sends the build description information to Watcher (4) so that it can perform the initial load-time weaving.

Watcher loads the aspect rules XML file (5) that specifies the aspect source files and pointcuts. Watcher uses CSharpParser⁸ to parse the source files of the target program and the aspects into their abstract syntax tree representations (AST) (6). The ASTs are then merged according to the aspect rules by inlining the advice code (7). The woven source files are generated by CSharpParser from the woven AST and are

⁷ Patcher consists of about 25,000 lines of C++ code. Watcher and Phx.Morph are written in C# and consist of about 3,700 and 7,700 lines of code, respectively. These figures include comments and blank lines but not generated code.

⁸ <http://www.debreuil.com/CSharp>

given a “.weaved.cs” extension (8). Creation of the woven source file artifacts is required for full source-level debugging and to support the next step.

The file names for the woven source files are passed along with the compiler options to the .NET CodeDOM API for compilation (source code to byte code) (9). This creates the woven executable, App-1.exe, along with its debug information file, App-1.pdb (10). (A number is appended to the executable name to ensure uniqueness.) Watcher sends a message to Patcher telling it the path to the woven executable (11). Watcher then proceeds to wait for changes to the target program’s source files, aspect source files, or aspect rule file.

Using the Metadata API, Patcher reads the metadata and method addresses associated with the running application (12) and the woven executable (13). It uses an executable differencing algorithm to compare the two modules (14). If one or more methods have changed, e.g., because an aspect was woven in, Patcher detects this and uses the Profiler API to update the byte code (15).

Runtime Weaving. At some point while the woven application is running, the user can make a change to the aspect rule file to enable/disable a specific aspect, enable/disable all aspects, introduce a new aspect, or modify the weaving rules (pointcuts). This is the main functional benefit of dynamic AOP, but it also provides debugging benefits since it shortens the edit-compile-debug cycle [6]. It allows the programmer to interactively eliminate aspects from the running program to isolate the cause of a fault, or to introduce a new *debugging aspect*, for example, to help diagnose a problem. Wicca supports *aspect introduction* (P5) so even aspects that were unanticipated prior to running the program can be woven in. This is useful for testing and debugging scenarios.

In addition to changing the aspect rule file, the user can modify any C# source file of the target program or aspects, a feature called *runtime modification* (P6) or *edit-and-continue*. This enhances the interactive debugging experience because the programmer can try out fixes to the base or aspect code without restarting.

Regardless of whether a base source file, aspect source file, or aspect rule file was changed, Wicca will detect the change and reweave, recompile, and update the running program by performing the same steps that we described for load-time weaving.

Static Weaving. Although it is missing in Figure 2, partly because it is not integrated with the dynamic weaver, Wicca v1 also supports static weaving via our Phx.Morph tool. Phx.Morph is a static byte-code weaver for the .NET platform. The Phx.Morph weaving model is depicted by the dashed line in Figure 1. Its language model is similar to AspectDNG⁹. Aspects can be defined in any .NET language and are compiled by a regular compiler. As opposed to our dynamic weaver which uses an XML file for

⁹ <http://www.dotnetguru.biz/aspectdng>

specifying the aspect rules, Phx.Morph uses regular .NET attributes attached to the aspect class.

Phx.Morph is built on top of Phoenix, Microsoft's production-grade compilation and tools infrastructure.¹⁰ A unique capability afforded by Phoenix is the ability to update the debug information of a Windows program. We know of no other byte-code weaver for .NET that supports this. This allows Phx.Morph to produce woven programs that are more debuggable than programs woven by other .NET byte-code weavers (AspectDNG for example), thus upholding the idempotence property (P1), and paving the way for eventual support for full source-level debugging.

5.2 The Wicca Debugger

We made a simple extension to the Microsoft cordbg command-line debugger to support source-level debugging of applications that have been dynamically updated by Wicca. An extension was required because standard debuggers do not support changing the debug information file (PDB) associated with the application being debugged.

The Profiler API supports updating debug information when a method body is replaced. However, it only handles injection-type modifications to the original method. While this is adequate for the advice inlining weaving technique used by Wicca, it does not handle arbitrary code changes. Since we believe edit-and-continue is an important debugging feature, and we plan to support debug obliviousness in the future, we extended cordbg to create the Wicca debugger, *wdbg*. We discuss *wdbg* further in §6.3.

5.3 Limitations

A limitation of Wicca is that dynamic weaving currently requires source for both the base program and the aspects. It is more restrictive than other dynamic AOP weavers in that regard. We plan to fully integrate our binary weaver, Phx.Morph, with our dynamic weaver to support full source, partial source, and no source scenarios, allowing us to enjoy full source-level debugging whenever source is available.

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.

Finally, our dynamic weaver is only a prototype and 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. Furthermore, our

¹⁰ <http://research.microsoft.com/phoenix>

dynamic weaver currently only allows method bodies to be updated. In contrast, our binary weaver provides more AOP functionality but less debugging capability.

6 Evaluation

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

6.1 Experimental Setup

We will use an aspect that embodies the *Design by Contract* (DBC) [36] principle. DBC allows the programmer to make assertions [25] about the system, in the form of *preconditions*, *postconditions*, and *class invariants*. For example, in Listing 1 we have a stack class for holding non-null elements. Its class invariant is that if the stack is nonempty, the top element 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. [29] Moreover, unlike normal assertions which are only checked for debug builds, or which require continuous checking at runtime, Wicca can inject these *test probes* [25] 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

```

public class StackDBCAspect {
    static int __savedCount;

    static void PreCond_push(Stack __this, object arg1) {
        if (arg1 == null)
            throw new ArgumentException("Precondition violated: Argument
                cannot be null", arg1);
        int savedCount = __this.count;
    }

    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

6.2 Detecting Faults using Test Probes

To test the stack class we created a test driver, StackDriver.exe, that pushes several items onto the stack and then pops each one while writing its value to the console.

When we launch StackDriver.exe, Wicca automatically attaches to it. Shortly after, we notice something odd: there appear to be duplicate items in the stack. While StackDriver is running, we enable the stack DBC aspect (*dynamism*), which may already exist or which we may have introduced for this debugging task (*aspect introduction*). Wicca detects this change and reparses, reweaves, and recompiles StackDriver.exe (*dynamism*). The time from the programmer saving the aspect rules file to Wicca updating StackDriver.exe is about 610ms.

Here is a snippet of the AspectRules.xml file after we added the stack DBC aspect and enabled 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>
  </aspect>
</aspects>

```

Immediately, the aspect code detects a postcondition violation and throws the exception: “Postcondition violated: Stack is empty after push.” The exception message

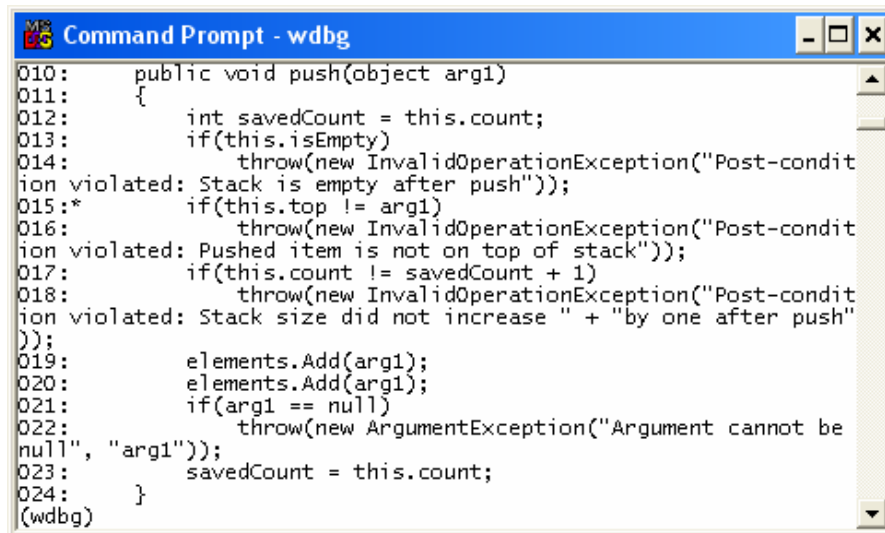
provides the file name and line number where the exception occurred. To figure out why, we launch wdbg, the Wicca debugger, and attach it to StackDriver.exe.

6.3 The Debugging Session

After we attach wdbg to StackDriver.exe we tell it to use the updated debug information file, StackDriver-1.pdb, that was created by the weave process (see §5.1). At this point, we can step into the push method and see the interwoven source code (see Figure 3). What is significant about this figure is that the base program and all AOP activities are debuggable at the source level (*debug intimacy*) — a first for a dynamic AOP system.

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 shown earlier 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.

A quick change to the aspect rules to fix this oversight causes Wicca to reparse, reweave, and recompile StackDriver. As expected, an exception is thrown immediately



```

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. 3. A wdbg debugging session showing aspect code interwoven with the stack class. This is the output immediately after attaching to the StackDriver.exe process and trapping the “Postcondition violated: Stack is empty after push” exception. The asterisk (*) indicates the current line

but this time with the correct message: “Postcondition violated: Stack size did not increase by one after push.”

We remove the extraneous Add method call by editing the source code in Stack.cs and saving Stack.cs, again causing Wicca to reparse, reweave, and recompile StackDriver (*edit-and-continue*). This time the program behavior looks correct and since no further violations are reported we conclude that the problem has been fixed.

At no time during the entire debugging session did we have to restart StackDriver.

6.4 Performance

The performance experiment was run on a Dell Dimension 8400 Workstation with a single Pentium IV 3.6 GHz processor with HyperThreading, an 800 MHz front-side bus, and 2 GB of 533 MHz SDRAM. The platform was Windows XP with Service Pack 2 and the .NET Framework version 1.1.4322.

For our benchmarks we used the StackDriver program, consisting of 66 C# source lines, and Goblin¹¹, a 3D game written by one of the authors that consists of 15,600 C# source lines.

We measured the end-to-end *update latency*, which is the time it takes to weave and update the application at load-time or runtime, using an aspect that traces the beginning of every method execution. Table 3 breaks down the update latency into its constituent components, averaged over 10 trials (5 load-time and 5 runtime trials) for each program. **Table 3 shows that the end-to-end update latency is less than 2.5 seconds for a medium-sized program and less than 1 second for a small program**, which we consider fast enough to support dynamic AOP as well as interactive debugging and development. However, due to some bugs in the CSharpParser, our tracing aspect was woven into only 68 of the 1126 methods in Goblin (6%). We assume that obtaining complete method coverage may increase the latency by 1-2 seconds.

Table 3. Update latency for weaving, compiling and updating the StackDriver and Goblin applications. The average (arithmetic mean) time of each operation is listed in milliseconds along with its standard deviation.

Component	Milliseconds (StdDev)		Contribution	
	<i>StackDriver</i>	<i>Goblin</i>	<i>StackDriver</i>	<i>Goblin</i>
Weaving	259.4 (121.3)	826.5 (72.2)	42.5%	35.0%
Compilation	329.6 (37.2)	611.1 (68.1)	54.0%	25.8%
Patch Creation	18.7 (6.8)	828.1 (182.1)	3.1%	35.0%
Update	3.2 (6.7)	98.5 (94.9)	0.5%	4.2%
Total Latency	610.9 (142.3)	2364.2 (339.6)	100.00%	100.00%

¹¹ <http://www.cs.columbia.edu/~eady/goblin>

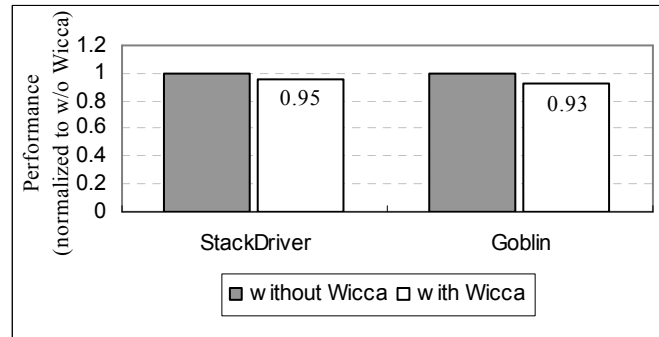


Fig. 4. Performance with and without Wicca enabled

Table 3 indicates that the majority of the time is spent weaving and compiling. They scale well, however, because the weaving time increases by a factor of 4 and the compile time increases by a factor of 2 when code size increases 200 fold. Patch creation does not scale as well as code size increases (it increases by a factor of 44). As Wicca is relatively unoptimized, we expect to be able to improve the efficiency of the weaving, compiling, and patch creation steps.

We determined the *runtime overhead* by measuring the time it takes Goblin to render 10,000 frames and for StackDriver to perform 140,000,000 stack push and pop operations averaged over 10 trials, running standalone and with Wicca attached. As this test was designed to measure the steady-state runtime overhead, no weaving or updating was performed. **Table 4 shows a runtime overhead of 5-7%.** We consider this an acceptable price to pay for enabling dynamic AOP and edit-and-continue functionality with support for full source-level debugging.

[13] contains further analysis of performance and possible optimization strategies.

7 Related Work

Similar to Wicca, SourceWeave.NET [28] parses source files using custom parsers, performs weaving on the abstract syntax tree, generates woven source file artifacts, and uses CodeDOM to compile the woven files. Both projects share the goal of improving debugging. SourceWeave.NET even chooses an aspect deployment model (*per-method*) believed to make aspect-related code easier to debug. Our main advance is that Wicca supports dynamic AOP allowing it to provide more debugging functionality (properties P4, P5, and P5). Replacing our front-end with SourceWeave.NET would allow us to weave using multiple languages, and would be a synergistic combination of the two projects.

The SELF compiler [26] shares our goal of providing an interactive development and debugging experience to the user without sacrificing performance. While Wicca supports optimized compiling of the woven code, *accurate* source-level debugging requires compiler optimizations to be disabled.¹² As we described in §4, this is a standard debugger requirement for debugging optimized code. The Profiler API used by Wicca imposes an additional restriction that requires method inlining to be disabled in the just-in-time compiler. SELF, on the other hand, is able to *deoptimize* methods on demand. If performance becomes an issue we will consider supporting deoptimization in Wicca as well.

8 Discussion

Wicca adds complexity to the weaving pipeline because it requires additional parse, compile, and update steps. Other dynamic AOP systems use the more direct approach of modifying the byte code or machine code in memory. After all, why create source when you can operate directly on the binary code? The main benefit of our approach is that it allows us to support source-level debugging and edit-and-continue which have not been top priorities for most AOP systems. Moreover, we have shown that this functionality can be provided efficiently, despite contrary statement in the literature.

8.1 Supporting Debug Obliviousness and Fault Isolation

Few AOP systems support debug obliviousness or fault isolation. To do this, the debugger must be able to identify AOP activity code, possibly using annotations (T4) or static analysis (T6). AspectJ and Steamloom support byte-code annotations for identifying aspects to prevent recursion during weaving [24] and to facilitate aspect removal [22]. As far as we know, no AOP system uses byte-code annotations to support obliviousness or fault isolation. We plan to explore this idea in Wicca.

A feature of debug information on Windows is that if the source line number associated with a region of byte code (or machine code) is set to 0xFEEFEE, the debugger will logically skip over the code, i.e., the code will still run but be hidden from the programmer. The *#line hidden* pragma can be used in C# source code to achieve the same effect. Although originally designed so that debuggers automatically “step over“ library or generated source code, we plan to repurpose these annotation-like mechanisms to support debug obliviousness in Wicca. Moreover, by adding a *#region*

¹² This is currently not a real limitation since the .NET Framework version 1.1.4322 does not perform any compiler optimizations.

pragma to aspect source code, the programmer can remain partially oblivious by choosing to show only the region name in the text editor.

8.2 Source-Level Debugging of Injected Code

All binary weavers, including Phx.Morph, have the problem that injected code cannot be debugged at the source level [2, 28], thus violating the debug intimacy property. The reason is the code may have been synthesized (i.e., the instructions were created on the fly) or imported from another module.

For example, AspectJ synthesizes and injects aspect selection, instantiation, and activation code, and some advice execution code, into the base program. In the debugger, the injected code is not visible in the source-level view at all, but can be seen by looking at the byte code. Other advice execution code is fully debuggable at the source level, however, due to AspectJ's implementation of the new Java specification for debugging support for different languages¹³. To support source-level debugging, the debugger can synthesize source on demand by reverse engineering (T5) AOP activity code [5]. Here again annotations can inform the process.

8.3 Arguments against Source-Level Debugging

Wicca supports the semantic model of AOP using source and binary code weaving. From the literature, these two AOP implementation techniques appear to be the most popular. We have shown that intimate source-level debugging is useful for weaver-based solutions because it allows the programmer to debug several types of AOP-specific faults as well as the weaving process itself. The ability to detect faults in the weaver is helpful when the weaving technology is immature.

While Wicca uses a somewhat radical approach, i.e., dynamic source weaving, to enable source-level debugging, this is merely an implementation detail. Other more efficient and direct ways of supporting source-level debugging of AOP-related code exist, which we touched upon briefly in this section.

Some AOP implementations may use runtime interception or a custom runtime environment and may not weave code at all. For these AOP implementations, the ability to debug AOP-related code at the source level does not make sense. However, these systems can still provide support for debug obliviousness and debug intimacy using some other means. For example, debug intimacy can be supported by showing a runtime visualization of the base program and aspect behavior. For debug obliviousness, only the base program behavior is shown.

¹³<http://jcp.org/aboutJava/communityprocess/final/jsr045>

9 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 fruitful results from that community applies to debugging woven code.

We demonstrated our Wicca system that advances the state of the art for debugging AOP. Wicca is the first dynamic AOP system to support full source-level debugging. It does this by employing a novel *dynamic source weaver* that combines source weaving with online byte-code patching with relatively low overhead. We created the first binary weaver for .NET to preserve debug information during weaving.

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 and Rean Griffith for their feedback. This research is funded in part by a grant from Microsoft Research.

References

- [1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, 1986.
- [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., March 2004.
- [3] S. Ausmann and M. Haupt. Axon – Dynamic AOP through Runtime Inspection and Monitoring. 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. Proc. of Prog. Language Design and Impl. (PLDI'05), June 2005.
- [5] J. Van Baalen, P. Robinson, M. Lowry, T. Pressburger. Explaining Synthesized Software. Proc. of Automated Software Eng. (ASE'98), October 1998.
- [6] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. Proc. of Aspect-Oriented Software Dev. (AOSD'02), April 2002.
- [7] C. Bockisch, M. Haupt, M. Mezini and K. Ostermann. Virtual Machine Support for Dynamic Join Points. Proc. of Aspect-Oriented Software Dev. (AOSD'04), March 2004.

- [8] J. Bonér. AspectWerkz — dynamic AOP for Java. Invited talk at the Conf. on Aspect-Oriented Software Dev. (AOSD'04), March 2004.
- [9] M. Ceccato, P. Tonella and F. Ricca. Is AOP code easier or harder to test than OOP code? Proc. of Testing Aspect-Oriented Programs (WTAOP 2005), March 2005.
- [10] A. van Deursen, P. Klint, and F. Tip. *Origin Tracking*. Journal of Symbolic Computation 15(5-6):523-545, May 1993.
- [11] R. Douence, T. Fritz, N. Lorient, J.-M. Menaud, M. Segura-Devillechaise and M. Südholt. An expressive aspect language for system applications with Arachne. Proc. of Aspect-Oriented Software Dev. (AOSD'05), March 2005.
- [12] R. Douence and M. Südholt. A Model and a Tool for Event-Based Aspect-Oriented Prog. (EAOP). Tech Rep 02/11/INFO. Ecole des Mines de Nantes, 2002.
- [13] M. Eaddy and S. Feiner. Multi-Language Edit-and-Continue for the Masses. Tech Rep CUCS-015-05. Dept. of CS, Columbia Univ., April 2005.
- [14] T. Elrad, R. Filman, and A. Bader. *Aspect-oriented programming: Introduction*. Communications of the ACM. 44(10): p. 29-32, 2001.
- [15] R. Faith. Debugging Programs after Structure-Changing Transformation. Ph.D. dissertation, CS Dept., Univ. of North Carolina, December 1997.
- [16] R. Filman and D. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. OOPSLA Wkshp. on Advanced Separation of Concerns. October 2000.
- [17] R. Filman and K. Havelund. Source-Code Instrumentation and Quantification of Events. Wkshp. on Foundations of Aspect-Oriented Languages (FOAL'02), April 2002.
- [18] R. B. Findler, M. Latendresse, and M. Felleisen. Object-oriented programming languages need well-founded contracts. Tech Rep TR01-372, Dept. of CS, Rice Univ., January 2001.
- [19] A. Frei, P. Grawehr, and G. Alonso. A Dynamic AOP-Engine for .NET. Tech Rep 445. Dept. of CS, ETH Zürich, March 2004.
- [20] W. G. Griswold, J. Yuan, and Y. Kato. Exploiting the Map Metaphor in a Tool for Software Evolution. Proc. of Software Eng. (ICSE '01), May 2001.
- [21] M. Haupt and M. Mezini. Micro-Measurements for Dynamic Aspect-Oriented Systems. Proc. of Net.ObjectDays. Springer-Verlag LNCS 3263, pp. 81-96, September 2004.
- [22] 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.
- [23] J. Hennessy. *Symbolic Debugging of Optimized Code*. ACM Transactions on Prog. Languages and Systems, 4(3): 323-344, July 1982.
- [24] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. Proc. of Aspect-Oriented Software Dev. (AOSD'04), March 2004.
- [25] C. A. R. Hoare. *Assertions: a personal perspective*. *Software pioneers: contributions to software engineering*, Springer-Verlag, pp. 356-366, 2002.
- [26] U. Hölzle, C. Chambers, and D. Ungar. Debugging Optimized Code with Dynamic Deoptimization. Proc. of Prog. Language Design and Impl. (PLDI'05), July 1992.
- [27] J. Hugunin. The next steps for aspect-oriented programming languages (in Java). Wkshp. on New Visions for Software Design & Prod.: Rsch. & Apps., December 2001.
- [28] A. Jackson and S. Clarke. SourceWeave.NET: Source-level cross-language aspect-oriented programming. Proc. of Generative Prog. and Component Eng. (GPCE'04), October 2004.
- [29] M. Lippert and C. V. Lopes. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. Proc. ICSE 2000, June 2000.

- [30] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, Proc. ECOOP 2001, Springer-Verlag LNCS 2072, pp. 327-353, Berlin, June 2001.
- [31] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. Tech Rep SPL97-008 P9710042, Xerox PARC, February 1997.
- [32] P. Klint, T. van der Storm, and J. Vinju. Term rewriting meets aspect-oriented programming. Tech Rep SEN-E0421, CWI, December 2004.
- [33] S. Krishnamurthi, M. Felleisen, and B. F. Duba. From macros to reusable generative programming. Proc. of Gen. and Comp.-Based Soft. Eng. (GCSE'1999), September 1999.
- [34] N. Kumar, B. Childers and M. L. Soffa. Tdb: a source-level debugger for dynamically translated programs. Proc. of Auto. and Analy.-Driven Dbg. (AADEBUG'05), Sept. 2005.
- [35] J. Lam. CLAW: Cross-Language Load-Time Aspect Weaving on Microsoft's Common Language Runtime, Demo at Aspect-Oriented Software Dev. (AOSD'02), April 2002.
- [36] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall, New Jersey, 1997.
- [37] A. Nicoara and G. Alonso. Dynamic AOP with PROSE. Wkshp. on Adaptive and Self-Managing Enterprise Applications (ASMEA'05), June 2005.
- [38] S. Redwine and W. Riddle. Software technology maturation. Proc. of Software Eng. (SE'85), August 1985.
- [39] 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.
- [40] F. Tip. Generic techniques for source-level debugging and dynamic program slicing. Proc. of Thy. and Prac. of Soft. Dev. (TAPSOFT'95), Springer-Verlag LNCS 915, May 1995.
- [41] 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.