Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy*

Curtis Clifton and Gary T. Leavens
Department of Computer Science, Iowa State University
226 Atanasoff Hall, Ames, IA 50011-1041 USA
+1-515-294-1580 {cclifton,leavens}@cs.iastate.edu

March 1, 2003

Abstract

The obliviousness property of AspectJ conflicts with the ability to reason about an AspectJ program in a modular fashion. This makes debugging and maintenance difficult. In object-oriented programming, the discipline of behavioral subtyping allows one to reason about programs modularly, despite the somewhat oblivious nature of dynamic binding; however, it is not clear what discipline would help AspectJ programmers obtain modular reasoning. We describe this problem in detail, and sketch a solution that allows programmers to tell both “superimposition” and “evolution” stories in their AspectJ programs.

1 Introduction

Much of the work on aspect-oriented programming languages refers to the work of Parnas [18]. Parnas argues that the modules into which a system is decomposed should provide benefits in three areas (p. 1054):

The benefits expected of modular programming are: (1) managerial—development time should be shortened because separate groups would work on each module with little need for communication; (2) product flexibility—it should be possible to make drastic changes to one module without a need to change others; (3) comprehensibility—it should be possible to study the system one module at a time. The whole system can therefore be better designed because it is better understood.

While much has been written about aspect-oriented programming as it relates to Parnas’s second point, his third point is the primary concern of this paper. AspectJ [2, 9] does not provide this third benefit in general, because it requires that systems to be studied in their entirety.

The whole-program analysis required by AspectJ is due to the form of obliviousness allowed by the language. Filman and Friedman define obliviousness as the execution of additional aspect-oriented code, A, without effort by the programmer of the client code that A cross-cuts [8]. Filman and Friedman’s paper argues that obliviousness is an essential property of aspect-oriented programming languages. But because of the obliviousness of AspectJ, it is difficult for programmers to find the code that may be executed at any given point in the program. This presents difficulties for debugging and for code understanding. This problem is currently being addressed via tool support (e.g., in Eclipse1, the QJBrowser2, and plug-ins for various IDEs) instead of via the language. Such tools perform the necessary whole-program analysis to direct the programmer to the applicable aspects that affect pieces of a module’s source code. Furthermore, the need for such tools indicates that client programmers cannot really remain oblivious to code introduced by aspects, except in a narrow technical sense.

It is interesting to examine the conflicting demands of comprehensibility and obliviousness in the context of object-oriented programming. Filman and Friedman point out that dynamic binding in object-oriented languages represents a form of obliviousness [8, §2]. Behavioral subtyping [7, 11, 12] restores Parnas’s ideal of comprehensibility to object-

---

*Copyright © 2003, Curtis Clifton and Gary T. Leavens, All Rights Reserved

1Information on the AspectJ development tools for Eclipse is available from http://www.eclipse.org/ajdt/
oriented programs, in spite of the obliviousness offered by dynamic binding. We will use this as an analogy for considering the problem of comprehensibility and obliviousness in AspectJ.

As an initial consideration of comprehensibility with aspect-oriented obliviousness, in this proposal we focus on AspectJ. We believe that our discussion applies to other AspectJ-like languages (such as AspectC). However, the problems and possible solutions we discuss do not necessarily apply to other aspect-oriented languages.

2 The Goal: Modular Reasoning

We would like to have both obliviousness and comprehensibility in an AspectJ-like language. This comprehensibility is often termed “modular reasoning”. Thus, we begin by defining a notion of modular reasoning that corresponds to Parnas’s comprehensibility benefit.

Modular reasoning is the process of understanding a system one module at a time. A language supports modular reasoning if the actions of a module M written in that language can be understood based solely on the code contained in M and the code surrounding M, along with the signature and behavior of any modules referred to by that code. The notion of a module and the surrounding code for a module are determined by the programming language. In Java, we might consider a compilation unit to be a module, and in standard Java no code surrounds a module. Code, C, refers to a module N if C explicitly names N, if C is lexically nested within N, or if N is a standard module in a fixed place (such as Object in Java).

We use contracts to specify the code’s behavior. In the most concrete case, the contract is the method’s code, but we prefer to think of more abstract contracts. These can be written in a formal specification language [10, 13], or informally by writing comments such as “This method returns true if the given file exists”.

Our interest in modular reasoning in aspect-oriented programming languages is motivated in part by our initial work on combining MultiJava [4, 6] and JML [10].

3 The Problem: Undisciplined Obliviousness

In typical object-oriented languages, the dynamic type of the receiver object is used to select the appropriate method to invoke for a given message send. Such dynamic selection of the target method can prevent modular reasoning. For example, consider the declaration of Point in Figure 1 and its method, move. The //@-comments before move’s declaration give its behavioral specification in JML.

- The clause “requires true” says that clients are not obliged to establish any precondition.
- The clause “modifies pos” says that the pos field of the receiver object may be changed by the method, but not any other locations.
- The clause “ensures ...” says that, after move returns, the value returned by newPos() is equal to the sum of the dist argument and the value returned by newPos() before move was called.

Suppose an object of static type Point is passed to a method client, as in Figure 2. If modular rea-
public class RightMovingPoint
    extends Point {
    public void move(int dist) {
        if (dist < 0) super.move(-dist);
        else super.move(dist);
    }
}

Figure 3: A RightMovingPoint class

reasoning is sound, then the programmer can reason about the invocation of move based on its specification in Point. That is, if the first assertion in Figure 2 holds, then the second assertion is valid based on the specification of Point’s move method. The definition of modular reasoning requires that the programmer should not have to consider possible subtypes of Point when doing this reasoning, since they are not mentioned in the client code. However, by subsumption, an instance of an unseen subtype of Point, say RightMovingPoint, may be passed to client. What if (as in Figure 3) RightMovingPoint overrides method move, but the override does not satisfy the specification of move in Point? Then modular reasoning such as that described for client is not valid. Using Figure 3, after the client’s invocation of p.move(-10), p.getPos() returns 10, not -10 as asserted.

The concept of behavioral subtyping restores sound modular reasoning by imposing the specification of Point on all its subtypes [7, 11, 12]. Thus, RightMovingPoint is not a behavioral subtype of Point, because its implementation does not satisfy the specification of move in Point. Behavioral subtyping is often described by saying that the behavior of a subtype should not be surprising, with respect to the specified behavior of a supertype.

As pointed out by Filman and Friedman [8], subtyping with subsumption, as above, is an example of obliviousness. Aspect-oriented programming languages allow programmers much greater latitude in defining oblivious behaviors.

3.1 Non-modular Reasoning in AspectJ

Next we show that, just as modular reasoning is not a general property of object-oriented programming languages in the absence of behavioral subtyping, modular reasoning is not a general property of AspectJ. We do this by considering an aspect-oriented extension to our Point example.

Figure 4 gives an aspect, MoveLimiting, that modifies the behavior of Point instances in a manner similar to that of RightMovingPoint. MoveLimiting declares a piece of around-advice. This advice intercepts calls to Point’s move method. If the argument passed to the client is negative, then, just as in RightMovingPoint, control proceeds to Point’s move method with the parameter set to the absolute value of the original parameter. As with RightMovingPoint, the client programmer’s reasoning, as indicated by the assert statements in Figure 2, is not correct in the presence of the MoveLimiting aspect.

In AspectJ the advice is applied by the compiler without explicit reference to the aspect from either the Point module or a client module. (Instead the classes and aspect are simply passed as arguments to the same compiler invocation.) Thus, modular reasoning about the Point module or a client module has no way to detect that the behavior of the move method will be changed when the Point module and MoveLimiting are compiled together. In AspectJ the programmer must potentially consider every combination of such aspects and the Point class in order to reason about the Point module. Some potentially applicable aspects may not even name Point directly, but instead may use wild card type patterns. So, in general, a programmer cannot “study the system one module at a time” [18].

Therefore, just as in object-oriented programming without behavioral subtypes, the obliviousness of AspectJ can prevent modular reasoning.

3.2 Problem Summary

The obliviousness of dynamic binding in object-oriented programming languages can prevent modular reasoning. The concept of behavioral subtyping was based on programmers’ stories about how they used and reasoned about subtypes. The formalization of behavioral subtyping builds on a programming discipline that evolved “in the wild”.

public aspect MoveLimiting {
    void around(int dist):
        call(void Point.move(int))
        && args(dist)
        {
            if (dist < 0) proceed(-dist);
            else proceed(dist);
        }
}

Figure 4: A MoveLimiting aspect
It is not yet clear how to modularly reason about AspectJ programs. What programming discipline, akin to behavioral subtyping, could be used to allow modular reasoning for AspectJ programs, while retaining obliviousness?

It seems that any such discipline must allow programmers to tell at least two sorts of stories:\footnote{Thanks to Arno Schmidmeier, Juri Memmert, Karl Lieberherr, Frank Sauer, and others for discussions at AOSD ’02 on the ways they are using aspect-oriented programming.}

- **superimposition stories**, combining modules for separate concerns without surprising behavior, and
- **evolution stories**, modifying the behavior of existing programs without changing the code of those programs.

4 Solution Sketch

Our solution sketch is based on separate approaches to telling superimposition and evolution stories.

4.1 Spectators

Superimposition stories are analogous to the stories told with subtypes in object-oriented languages; both seek to enhance existing behavior without introducing surprising behavior. We hope to exploit this analogy to understand how to restore modular reasoning for AspectJ, while retaining obliviousness. Following this analogy we seek additional constraints on aspects.

We propose a category of aspects, called **spectators**, that are prevented, in some well-defined sense, from changing the behavior of the modules that they advise. Spectators will allow programmers to tell superimposition stories.

The term “spectator” is intended to denote the hands-off nature of these aspects. We consider two possible interpretations for this notion.

- We could limit the state that may be mutated by a spectator to just that state that it owns (in the sense of any of the various ownership type systems \[1, 14, 15, 16\]). Such a limitation is statically checkable, but requires additional annotations in the source code. However, these annotation offer the additional benefit of controlling aliasing.
- We could allow spectators to do anything that does not violate the specification of the advised modules. Such an approach gives broad latitude to the spectator’s programmer, but requires verification techniques beyond simple static checking (e.g., theorem proving). Also, since aspects provide advice via pattern matching (that is, they use quantification \[8\]), it is not clear that one could modularly find all the modules advised by a given spectator, which is necessary to determine the set of inviolable specifications.

In addition to these safety properties, we must also consider liveness properties. For example, must advice in spectators always proceed to the advised method? It seems like that should be the case, but because of the halting problem, checking this is not decidable in general. Possible solutions include:

- restricting the set of control flow constructs allowed in spectators, so that the checks are decidable,
- having spectator advice run in a separate thread, so that the main thread of control always continues to the advised method, or
- prohibiting spectators from using around-advice or throwing checked exceptions, so that control proceeds in normal circumstances.

4.2 Assistants

To tell evolution stories, programmers need a category of aspect that allows them to change the behavior of existing modules. We call such aspects **assistants** to denote their active role. We propose that assistants should have the full expressive power of AspectJ aspects. The **MoveLimiting** aspect of Figure 4 would be a valid assistant, but we have already seen that such aspects prevent modular reasoning. So how do we reconcile assistants with our desire for modular reasoning?

We think the answer is that we must limit the obliviousness of client code with regard to assistants. We propose that an AspectJ-like language include a module interconnect language that allows programmers to specify which assistants are potentially applicable to any given client module. (The filter specifications in Composition Filters \[3\] have this flavor, as does Hyper/J’s meta-language for compositing hyperslices into hypermodules \[19, 17\], perhaps indicating that these languages are better suited for modular aspect-oriented reasoning.)

The module interconnection specifications for a given client module would be found in a well-defined
place relative to the client module. A single inter-
connection specification might describe the intercon-
nections for multiple client modules, so that pattern
matching could still be used. Because the inter-
connection specification would be in a well-defined
place, the reader of a client module could readily
identify all of the potentially applicable assistants
for that client module. This ready identification al-
ows modular reasoning and obviates the need for a
whole-program analysis.

In our discussions with users of AspectJ we found
that most are using their build systems (e.g., Ant or
Make) in place of the proposed module interconnect-
language. We are suggesting that, instead of using
ad hoc techniques to specify the interconnection of
aspects and types, such interconnection should be
part of the language.

5 Summary and Future Work

We have shown that obliviousness can prevent mod-
ular reasoning, as demonstrated by the MoveLim-
iting aspect, and by the (non-behavioral) subtype
RightMovingPoint. Behavioral subtyping adds
modular reasoning to object-oriented programming
languages. We seek a similar programming dis-

cipline for AspectJ-like languages that will allow
sound modular reasoning while still letting program-
ners tell superimposition and evolution stories.

We have suggested that, appropriately defined,
spectators and assistants might add modular rea-
soning to AspectJ-like languages. Spectators in
an AspectJ-like language could be used to tell su-
perimposition stories. Just as behavioral subtyp-
ing places constraints on subtype designers so that
client programmers may remain oblivious, the de-

inition of spectators constrains aspect designers so
that client programmers may remain oblivious. As-
sistants, together with a module interconnection lan-
guage, could be used to tell evolution stories. The
cost of telling evolution stories is that client pro-
grams may not be completely oblivious to the mod-
ule interconnections.

It is tempting to relate the superimposition and evolu-
tion stories to the distinction between produc-
tion and development aspects made by others [9].
However, it seems that there are differences. For
example, distinct production aspects for persistence
and accounting rules might be part of a superim-
position story if their specifications do not interact.
We hope to better understand these distinctions by
investigating possible solutions to the problem pre-

We plan to formalize the problem and investigate
the solution space by building a core calculus for
an AspectJ-like language with support for modular
reasoning. After proving appropriate modularity
properties for this calculus we will develop a
full-scale programming language (a successor to As-
pectJ) with similar properties.

Another avenue for understanding possible solu-
tions is to investigate the modular reasoning prop-
erties of aspect-oriented languages other than As-
pectJ.

The parallel of obliviousness in subtyping and in
aspect-oriented programming, pointed out by Fil-
man and Friedman [8] provokes a useful analogy be-
tween reasoning in the two styles. It may be inter-
esting to consider how the notions of spectators and
assistants might provide new insight on the earlier
work on behavioral subtyping.

6 Acknowledgments

We thank Yoonsik Cheon, Todd Millstein, Markus
Lumpe, and Robyn Lutz, for their helpful com-
ments on an early version of this work [5]. We
also thank the workshop participants at Founda-
tions of Aspect-oriented Languages 2002, in partic-
ular Gregor Kiczales, Doug Orleans, and Hidehiko
Masushara, for discussions regarding this work. Fi-
nally, we thank the anonymous referees from the
Aspect-Oriented Software Development 2003 Pro-
gram Committee, for helping us to understand the
scope of the problems we are addressing, and from
Software-Engineering Properties of Languages for
Aspect Technologies 2003, for helping us tighten the
presentation in this paper.

The work of both authors was supported in part
by a grant from the US National Science Foundation
under grants CCR-0097907 and CCR-0113181.

References

Alias annotations for program understanding.
In Proceedings of the 17th ACM conference
on Object-oriented programming, systems, lan-
guages, and applications, volume 37(11) of
ACM SIGPLAN Notices, pages 311–330. ACM,
Nov. 2002.


