A Classification System and Analysis for Aspect-Oriented Programs

Martin Rinard, Alexandru Sălcianu, and Suhabe Bugrara
Massachusetts Institute of Technology
Cambridge, MA 02139

ABSTRACT

We present a new classification system for aspect-oriented programs. This system characterizes the interactions between aspects and methods and identifies classes of interactions that enable modular reasoning about the crosscut program. We argue that this system can help developers structure their understanding of aspect-oriented programs and promotes their ability to reason productively about the consequences of crosscutting a program with a given aspect.

We have designed and implemented a program analysis system that automatically classifies interactions between aspects and methods and have applied this analysis to a set of benchmark programs. We found that our analysis is able to 1) identify interactions with desirable properties (such as lack of interference), 2) identify potentially problematic interactions (such as interference caused by the aspect and the method both writing the same field), and 3) direct the developer's attention to the causes of such interactions.

Categories and Subject Descriptors: D.2.4 [Software/Program Verification]: Validation, D.3 [Software]: Programming Languages

General Terms: Languages, Verification.

Keywords: Aspect oriented programming, program analysis.

1. INTRODUCTION

Aspect-oriented programming languages enable the isolation of crosscutting concerns in aspects, with the advice in these aspects invoked at the appropriate points in the execution of the program [15, 21, 19, 2]. This mechanism supports the development of programs whose structure (a core code base combined with crosscutting aspects) more closely corresponds to their designs (which often separate crosscutting concerns from the conceptual core of the system).

*This research was supported in part by DARPA Contract F33615-00-C-1692, NSF Grants CCR00-86154, CCR00-63513, and CCR02-09075, and the Singapore-MIT Alliance.

SIGSOFT'04/FSE-12, Oct. 31–Nov. 6, 2004, Newport Beach, CA, USA.

The mechanisms in aspect-oriented languages support a broad range of composition strategies, from the clearly acceptable (adding logging code to method invocations) to the questionable (exchanging the values of object fields prior to the execution of a method that accesses the object). Depending on the way it is used, aspect-oriented programming can either simplify the structure of the program (with a corresponding simplification in the reasoning required to understand its structure and behavior) or destroy its conceptual integrity (making it impossible to reason in a modular way about the crosscut program). The behavior of the final system depends on the interaction between the aspects and the core program; the fundamental issue is that some kinds of interactions support modular reasoning while others actively interfere with it.

This paper presents a new classification system for aspect-oriented programs. We have three goals:

- **Characterization:** To precisely identify interaction patterns that support certain kinds of modular reasoning.
- **Automatic Recognition:** To develop a program analysis algorithm that is capable of automatically recognizing these interaction patterns.
- **Focused Attention:** If the interaction pattern does not fall into a category known to support modular reasoning, to focus the attention of the developer on the particular elements of the interaction that prevent it from doing so.

We have implemented this analysis to provide developers with a system that may help them understand the interactions in their aspect-oriented programs. By automatically recognizing interaction patterns known to support modular reasoning, our system helps the developer understand the consequences of crosscutting the program with a given aspect and provides a guarantee that this understanding is justified. If the interaction between the aspect and the program does not fit any of these patterns, the system can direct the developer's attention to the cause of the mismatch. The developer can then explore this cause further with the confidence of a guarantee that this cause is the only potential issue.

1.1 Classification System

Because aspects crosscut programs at the granularity of advice, our classification system focuses on interactions between advice and methods. We address two kinds of interactions: direct interactions between an advice and methods that it crosscuts and indirect interactions between an advice and methods that may access the same object fields.[1]

Our classification system for direct interactions focuses on control flow elements that affect how and when a crosscut method executes:

- **Augmentation:** After crosscutting, the entire body of the method always executes. Advice with functionality orthogonal to the functionality of the method (for example, logging and monitoring advice [17]) is typically augmentation advice.
- **Narrowing:** After crosscutting, either the entire body of the method executes or none of the body executes — in effect, the advice conditionally executes the method. Advice that checks safety conditions (such as security or consistency conditions) before allowing the method to execute is often narrowing advice.
- **Replacement:** After crosscutting, the method does not execute at all — the advice replaces the behavior of the method with completely new behavior. Advice that uses a static pointcut specification to check a safety condition (rather than an explicit dynamic check like narrowing advice) is usually replacement advice. Because replacement advice completely eliminates part of the system, it may also indicate the presence of modifications that did not fit within the original design.
- **Combination:** After crosscutting, the method and aspect combine in some other way to produce potentially new behavior.

Note that this axis of our system classifies each advice in isolation — regardless of the methods that it may or may not crosscut, each advice is either an augmentation advice, a narrowing advice, a replacement advice, or a combination advice.

The other axis of our classification system associates a *scope* with each advice and method of the core software system. Each scope identifies, at the granularity of fields of classes, the part of the state that comprises the concern of the advice or method. So, for example, the scope of a collection of methods that maintain graphical objects for display (circles, squares, etc.) might be the fields that define the positions and sizes of the objects on the screen. Each scope specifies whether advice or methods in the scope may read or write each field.

To avoid exposing the internal representation of encapsulated data and to model externally visible side effects (such as printing a check or moving a control surface) scopes may also include *abstract fields*. Each abstract field either represents a collection of encapsulated fields or enables the system to summarize externally visible actions as reads or writes to abstract fields. Our classification system uses scopes to identify the following kinds of interactions:

- **Orthogonal:** The advice and method access disjoint fields. In this case we say that the scopes are *orthogonal*.
- **Independent:** Neither the advice nor the method may write a field that the other may read or write; in this case we say that the scopes are *independent*.
- **Observation:** The advice may read one or more fields that the method may write but they are otherwise independent. In this case we say that the advice scope *observes* the method scope.
- **Actuation:** The advice may write one or more fields that the method may read but they are otherwise independent. In this case we say that the advice scope *actuates* the method scope.
- **Interference:** The advice and method may write the same field; in this case we say that the two scopes *interfere*.

1.2 Reasoning

Developers of aspect-oriented programs may find themselves in a variety of situations: deploying a new aspect into an existing program, attempting to understand the structure and operation of a given aspect-oriented program, or developing a new aspect-oriented design. As we describe below, our classification system and analysis is designed to support the reasoning required for developers to perform effectively in these kinds of situations.

1.2.1 Deploying a New Aspect

A developer contemplating the deployment of a new aspect into an existing program must answer the following question: If I understand the program before I deploy the aspect, what do I have to do to understand the program after the aspect crosscuts it? Our system identifies situations in which the developer can simply reuse his or her original understanding in a modular way to understand the crosscut program. It also identifies when the developer must rethink his or her understanding of the behavior of parts of the original program in the context of their interaction with the aspect before he or she can obtain an accurate understanding.

One principle of our system is that using augmentation advice to introduce additional behavior in orthogonal, independent, or observation scopes supports modular reasoning because the aspects do not change the behavior of the program within its original scopes. The developer's unchanged understanding of the original program therefore remains completely valid when reasoning about the behavior of the crosscut program within these original scopes.

Using an aspect to disturb the operation of the program within its original scopes, on the other hand, complicates modular reasoning because it may change the encapsulated behavior of the methods in the program. In this case, the developer must go back and reason about how these changes propagate up the method call hierarchy and through the affected scopes before he or she can understand the crosscut program — clearly a much more involved task than understanding the consequences of deploying an aspect whose scopes are orthogonal to those of the original program.

Aspects with narrowing advice occupy a middle ground. Identifying the advice as narrowing advice enables the developer to focus his or her attention on the narrowing condition. In many cases this condition is designed to catch violations of a safety property; if the developer can check that the

[1]It is also possible for one piece of advice to crosscut other pieces of advice or to access the same fields as other pieces of advice. Our implemented system handles these kinds of *advice-on-advice* interactions. To simplify the presentation, however, we frame the discussion in terms of interactions between advice and methods.

condition always holds, his or her original understanding remains valid because the original and crosscut programs have the same behavior. Even if the condition may be violated, the developer can use the narrowing condition to focus on only those parts of the execution that it may affect.

1.2.2 Understanding Aspect-Oriented Programs

A developer attempting to understand an existing aspect-oriented program must answer the following two questions: How is the functionality partitioned into advice and methods, and how do the advice and methods interact to deliver the overall behavior of the program? Our system uses scopes to provide structural information about how the functionality is partitioned into advice and methods across the program. Because scopes summarize the parts of the state that each element accesses, they enable the developer to quickly find and focus on those advice and methods that may access a relevant part of the state. They also allow the developer to immediately discard those that do not.

Our system is also designed to provide behavioral information about how control flows through the advice and methods. Aspect-oriented programs contain two kinds of control flow: *explicit* control flow at method invocation sites, and *implicit* control flow at joint points that execute advice. Our classification system is designed to enable developers to quickly link together both kinds of control flow to understand how the advice and methods interact to deliver the overall behavior of the system. Augmentation advice has a quite simple effect on the control flow — it simply combines the execution of the advice and the crosscut method. Narrowing advice is a bit more complex because it requires the developer to reason about the narrowing condition. Replacement advice is in some sense the simplest — because it replaces the crosscut method with the execution of the advice, the developer does not need to consider the execution of the method at all. In general, combination advice is the most complex because it allows arbitrary combinations of the advice and crosscut method executions.

1.2.3 Structuring Aspect-Oriented Designs

Finally, a developer producing an aspect-oriented design must answer the following question: How should I best partition the functionality to group closely coupled interactions within related advice or methods and eliminate (or at least minimize) undesirable interactions between aspects and classes? In general, we expect appropriately modularized designs to decompose the functionality into largely orthogonal, independent, or observation advice and methods, with most of the advice either augmentation or narrowing advice. Interference or some forms of replacement or combination advice can be a sign of an inadequately modularized design and may therefore provide a signal to the developer to rethink this part of the design (although modular patterns that involve combination advice or interference interactions may emerge over time). As this discussion illustrates, the basic concepts in our classification system (scopes, advice classifications, and interaction classifications) provide a framework that designers can use to conceptualize the issues involved and, ideally, to drive their designs toward better modularity. Each advice or method should have a clearly identified concern (as realized in its scope); different concerns should interact, if at all, only in structured and modular ways.

1.3 Analysis

Using scopes to reason about an aspect-oriented program is clearly unsound if the advice and methods do not conform to their scopes. We therefore use a modified pointer and escape analysis to automatically extract a specification of the fields that the execution of each advice or method (including any transitively invoked methods) may access. A comparison of this extracted specification with the scope enables the analysis to determine if the advice or method performs only those accesses identified in the scope. The inclusion of escape information provides two benefits:

- **Captured Objects:** It enables the analysis to recognize captured objects whose lifetimes are included in the lifetime of the advice or method in which they are allocated. Because these objects are not visible outside the analyzed advice or method, the analysis omits any accesses to fields in these objects from the extracted access specification. The analysis therefore supports the use of common programming patterns (such as iterators) that allocate and use temporary objects. Without this feature, commonly used classes such as iterators and `StringBuffer`s would generate a large number of false Interference interactions.

- **New Objects:** It enables the analysis to distinguish accesses to new objects from accesses to previously existing objects, which in turn enables the analysis to recognize the absence of conflicts between accesses to an existing object and accesses to a new object. The analysis therefore supports programming patterns (such as storing composite return values in new objects) that allocate and use new objects.

This analysis gives the developer significant flexibility when obtaining the scopes. It is possible to simply use the extracted access specification, which completely eliminates any scope specification overhead. It is also possible to use the analysis to automatically derive some initial scopes, then abstract scopes as appropriate to obtain the final scopes. It is also possible to develop the scopes from scratch (this approach may be especially appropriate during design when the source code does not exist).

The analysis uses the scopes to classify indirect data interactions between methods and advice. It also performs a control-flow analysis of each advice in isolation to classify each advice as either augmentation advice, narrowing advice, replacement advice, or combination advice.

1.4 Contributions

This paper makes the following contributions:

- **Classification System:** It presents a classification system for aspect-oriented programs. This system characterizes both direct control flow interactions between advice and crosscut methods and indirect interactions that take place as the advice and methods access object fields.

- **Scopes:** It introduces the concept of scopes as a concrete representation of the concern of an advice or method. Each scope identifies the correspondence between the concern and accessed object fields; abstract fields hide encapsulated implementations and summarize actions with externally visible effects.

- **Analysis Algorithm:** It presents an implemented analysis algorithm that the developer can use to either 1) automatically extract scopes for advice and methods or 2) verify that advice and methods conform to their scopes.

- **Reasoning Support:** It discusses the reports that the analysis generates and shows how the developer can use these reports to identify interaction patterns that support modular reasoning. The reports also identify any interactions that may interfere with modular reasoning, enabling the developer to focus on these interactions when reasoning about the crosscut program.

- **Experience:** It presents our experience using our system to analyze several aspect-oriented programs. Our results show that our system can accurately classify many pieces of advice and methods into categories that support modular reasoning and precisely identify the cause of any potentially non-modular interaction.

2. EXAMPLE

We next present an example that illustrates how our analysis can help developers understand aspect-oriented programs. Figure 1 presents the code for a `Stack` of integers. The `Stack` class exports the standard `push(int)` and `pop()` methods; it also allows a client to obtain an iterator and iterate through the items in the stack. Figure 2 presents the code for the iterator.

2.1 A Consistency Checking Aspect

The stack comes with the consistency property that all items in the stack must be nonnegative. Figure 3 presents the `NonNegative` aspect, which contains advice that iterates through all of the items in the stack to check that this property holds. The pointcut in the advice specifies that it should run before any `Stack` method.

Our control flow analysis recognizes the `NonNegative` advice as narrowing advice — the consequence of crosscutting the method with this advice is that the method executes normally unless the stack fails the consistency check. The net effect is to narrow the set of input states on which the method will execute.

To enable the analysis of indirect interactions between the advice and the methods (these interactions take place when the advice and a method access the same object field), our system requires scopes for each advice and method. If desired, the combined pointer and escape analysis presented in Section 3 can automatically derive these scopes. Figure 4 presents the results of this analysis for our example.[2] We use the notation `Cell.Cell(Cell, int)` to denote the constructor of the `Cell` class, and similarly for other constructors. Because advice is anonymous, we identify each piece of advice with its position in the aspect (so `NonNegative 1` identifies the first advice in the `NonNegative` class). Each potential access is identified by the class and field (so `write Stack.head` identifies a write to the `head` field of a `Stack` object). We use the notation `write new Cell.next` to indicate that the write is occuring to a `Cell` object that did not exist when the advice or method was invoked.

Note that the scope of the advice does not include accesses to any of the fields of the `Iterator` object that it uses to iterate through the stack. The escape analysis recognizes that the lifetime of the iterator is contained in the lifetime of the advice execution. Because the scope is designed to summarize only the externally observable accesses, the analysis omits all of the accesses to these iterator fields.

The developer may be satisfied with the automatically extracted scopes presented in Figure 4. Our system does, however, enable the developer to specify an abstraction function that the system can apply to convert some (or even all) of

[2]We present the scopes for the classes in Figures 1 and 2 and the aspect in Figure 3. For this example, our implemented system also produces scopes for hundreds of methods from the Java standard libraries.

```
class Cell {
  Cell(Cell n, int i) { next = n; data = i; }
  int data; Cell next;
}

class Stack {
  Stack() { head = null; }
  Cell head;
  boolean push(int i) {
    if (i < 0) return false;
    head = new Cell(head, i);
    return true;
  }
  int pop() {
    if(head == null)
      throw new RuntimeException("empty");
    int result = head.data; head = head.next;
    return result;
  }
  Iterator iterator() { return new StackItr(head); }
}
```

Figure 1: Stack Class

```
interface Iterator {
    public boolean hasNext();
    public int next();
}

class StackItr implements Iterator {
  private Cell cell;
  public StackItr(Cell head) { this.cell = head; }
  public boolean hasNext() { return cell != null; }
  public int next() {
    int result = cell.data; cell = cell.next;
    return result;
  }
}
```

Figure 2: Stack Iterator

```
aspect NonNegative {
  before(Stack stack) : call(* Stack.*(..)) &&
                        target(stack) &&
                        !within(NonNegative) {
    Iterator it = stack.iterator();
    while(it.hasNext()) {
      int i = it.next();
      if(i < 0)
        throw new RuntimeException("negative");
    }
  }
}
```

Figure 3: Stack Consistency Checking Aspect

```
Cell.Cell(Cell, int) : write Cell.next, Cell.data;

Stack.Stack() : write Stack.head;
Stack.pop() :
   read Stack.head, Cell.next, Cell.data;
   write Stack.head,
         new Java.lang.Throwable.detailMessage;
Stack.iterator():
   read Stack.head;
   write new StackItr.cell;
Stack.push(int) :
   read Stack.head;
   write Stack.head, new Cell.next, new Cell.data;

StackItr.StackItr(Cell) : write StackItr.cell;
StackItr.hasNext() : read StackItr.cell;
StackItr.next() :
   read StackItr.cell, Cell.next, Cell.data;
   write StackItr.cell;

NonNegative 1 :
   read Stack.head, Cell.next, Cell.data;
   write new Java.lang.Throwable.detailMessage;
```

Figure 4: Automatically Extracted Scopes

```
field Stack.structure, StackItr.structure,
      Java.lang.Throwable.structure;

Cell.next -> Stack.structure;
Cell.data -> Stack.structure;
Stack.head -> Stack.structure;
StackItr.cell -> StackItr.structure;
Java.lang.Throwable.detailMessage ->
  Java.lang.Throwable.structure
```

Figure 5: Abstraction Function for Stack Example

```
Cell.Cell(Cell, int) : write Stack.structure;

Stack.Stack() : write Stack.structure;
Stack.push(int) :
   read Stack.structure;
   write Stack.structure;
Stack.pop() :
   read Stack.structure;
   write Stack.structure,
         new Java.lang.Throwable.structure;
Stack.iterator():
   read Stack.structure;
   write new StackItr.structure;

StackItr.StackItr(Cell) : write StackItr.structure;
StackItr.hasNext() : read StackItr.structure;
StackItr.next() :
   read StackItr.structure, Stack.structure;
   write StackItr.structure;

NonNegative 1 :
   read Stack.structure;
   write new Java.lang.Throwable.structure;
```

Figure 6: Scopes After Abstraction

```
NonNegative 1 (narrowing)

Crosscuts
  Stack.push(int) : observation
      read/write : Stack.structure;
      read/read : Stack.structure;
  Stack.pop() : observation
      read/write : Stack.structure;
      read/read : Stack.structure;
  Stack.iterator() : independent
      read/read : Stack.structure;

Interactions
  StackItr.next() : independent
      read/read : Stack.structure;
  Stack.Stack() : observation
      read/write : Stack.structure;
  Cell.Cell(Cell, int) : observation
      read/write : Stack.structure
```

Figure 7: Analysis Report for `NonNegative` Aspect

the concrete fields in the scopes into abstract fields. This mechanism may be useful for hiding the internal implementation details of various components such as the `Stack` in our example. Figure 5 presents one potential abstraction function in our example. It introduces a new abstract field, the `Stack.structure` field (which represents the fields in `Stack` data structure) and maps the `Stack.head`, `Cell.next`, and `Cell.data` fields to this field.

Figure 6 presents the new scopes after abstraction. All of the internal fields of the `Stack` data structure (including the helper class `Cell`) have been replaced by the single abstract field `Stack.structure`, and similarly for other classes with encapsulated fields.

Given these scopes and the result of the control-flow analysis, our system produces the report in Figure 7 that characterizes the interactions between the advice and the methods in the program. This report indicates that:

- **Narrowing:** The advice in the `NonNegative` aspect is narrowing advice.
- **Crosscuts:** The advice crosscuts the `Stack.push(int)`, `Stack.pop()`, and `Stack.iterator()` methods.
- **Observation Interactions:** The advice reads fields that the `Stack.Stack()`, `Stack.push(int)`, `Stack.pop()`, and `Cell.Cell(Cell, int)` methods write, but does not write fields that the methods access.
- **Independent Interactions:** The advice reads object fields that the `Stack.iterator()` and `StackItr.next()` methods also read.

Note that the advice and the `Stack.pop()` method both write the `Java.lang.Throwable.structure` abstract field, but because the writes access a new object in each case, there is no interaction.

We expect that a developer might use this report as follows. First, he or she would examine the object field interactions and find that the advice has at most observation interactions with the methods in the program. Therefore, the only way that the advice can change the operation of the program within its original set of scopes is to affect the flow of control. The developer would therefore inspect the narrowing condition in the advice to understand under what conditions the advice would prevent a crosscut method from executing. In this case, it is clear that the advice is designed to check that all elements in the stack are nonnegative; if the stack never violates this condition, the advice will not affect the execution of the program. The developer would then examine the `Stack` implementation, verify that the `Stack.push(int)` method refuses to insert negative elements into the stack, and conclude that his or her understanding of the original program is still valid in the crosscut program. In our view, the report offers two benefits in this example:

- **Eliminating Potential Issues:** It provides a guarantee that certain kinds of interactions (for example, the advice writing a field that the program subsequently reads) do not happen. This frees the developer from having to consider the possibility of these interactions when reasoning about how the advice may affect the program.

- **Focused Attention:** It focuses the attention of the developer on those parts of the implementation that he or she must examine to understand the consequences of deploying the aspect. In this example, the report focuses the attention of the developer on the narrowing condition and those parts of the program that may affect the narrowing condition.

2.2 An Instrumentation Aspect

Figure 8 presents a centralized `InstrumentationData` class that contains a count of the number of times the crosscut program invokes a `push` method (in any class). The advice in the `Instrumentation` aspect increments this count after every call to a `push` method.

After automatic scope extraction, our system produces the report in Figure 9. This report first indicates that the advice is an augmentation advice that does not affect the control flow of crosscut methods. It also indicates that the advice is orthogonal to every method in the program except the `InstrumentationData.print()` method (the remaining methods include methods from the `Stack`, `Cell`, and `StackItr` classes from our example). The method reads the `InstrumentationData.count` field; the advice writes this field. We note in passing that there is a write to the abstract field `java.lang.System.print` in the scope of the `InstrumentationData.print()` method. This abstract field write summarizes the externally visible action of printing.

We expect the developer to use this report as follows. An examination of the report quickly shows that the aspect can affect the execution of the original program only within scopes that include `InstrumentationData.count`, and the only potential interaction at that field occurs inside the `InstrumentationData.print()` method. An examination of this method indicates that this is an anticipated and desired interaction. The conclusion is that crosscutting the program with the aspect preserves the desired behavior and introduces no unanticipated interactions or control flow changes. Even though the aspect does not leave the behavior of the crosscut program intact within its scopes, the developer was able to use the report to quickly isolate any potential issues and verify that the interactions were all anticipated and desired.

```
public class InstrumentationData {
    static int count = 0;
    public print() {
        System.out.println("count = " + count);
    }
}

aspect Instrumentation {
    after() : call(* *.push(int)) {
        InstrumentationData.count++;
    }
}
```

Figure 8: `Instrumentation` Aspect

```
Instrumentation 1 (augmentation)

Crosscuts
  Stack.push(int) : orthogonal

Interactions
  InstrumentationData.print() : actuation
      write/read : InstrumentationData.count;
      read/read : InstrumentationData.count;
```

Figure 9: Analysis Report for `Instrumentation`

2.3 Developer-Provided Scopes

In this example we have focused on the use of automatically extracted scopes with the developer applying abstraction functions to hide internal implementation details. Our system also generalizes to support the use of scopes provided directly by the developer. In general, we expect such scopes to be much coarser than the automatically extracted scopes — specifically, we expect the developer to identify a relatively small number of scopes, then use the same scope for many classes or methods that share the same concern. Our system also supports multiple scopes for classes or methods with multiple concerns, enabling the developer to appropriately factor the scope space.

While providing these scopes may require some additional specification effort, this effort may be justified. First, we have found scopes to be a useful conceptual tool for reasoning about the concerns of the different parts of the program. Second, they enable the analysis to generate useful reports for partial programs that are still under construction — the report generator works with scopes only and does not require source code.

3. ANALYSIS

We next present the core analyses: the underlying pointer analysis, the extension of this analysis to record and generate information about accessed object fields, and the control flow analysis that recognizes augmentation, narrowing, replacement, and combination advice.

3.1 Analyzed Language

Our analysis is designed to work with languages (such as AspectJ [17]) with aspects that contain (one or more pieces of) advice. A *join point* is a point in the execution of the program (for example, the execution of a method). Each advice contains a *pointcut specification*, (which identifies a set of join points), a piece of code (which executes whenever control flow reaches one of these join points), and an advice kind (which specifies whether the code in the advice executes before, after, or around the join point). Around advice uses a *proceed* statement to execute the join point around which it executes.

Pointcut specifications may include both static parts (for example, the execution of a specific method or an access to a specific field) and dynamic parts (for example, an arbitrary boolean condition involving the parameters from the join point or a condition that specifies that the join point must be contained within the control flow of a given method). In general, resolving the dynamic part of the pointcut specification may require information that is available only as the program runs. Any static analysis that attempts to identify the set of join points that match a given pointcut specification must therefore use some abstraction to approximate the dynamic part of the pointcut specification.

Our current system is designed for method execution and method call join points only. Our direct interaction classification system, however, applies directly to any kind of join point. It is also possible to generalize our indirect interaction classification system to include other kinds of join points. For each kind of join point, the resulting classification would depend on the memory accesses that the join point could perform. For example, an appropriate indirect interaction classification system for field access join points would be based on the aspect and the join point's combined accesses to the field.

3.2 Pointer Analysis

Our analysis is based on an existing interprocedural, flow-sensitive, context-sensitive, bottom-up combined pointer and escape analysis [24]. We augmented this analysis to maintain additional information about the accesses that each advice and method performs and use this information to generate or verify the advice and method scopes.

Heap abstraction: The analysis uses points-to graphs to model the heap that the analyzed program accesses: the nodes of these graphs represent heap objects; the edges represent references in the heap. Conceptually, the analysis computes one points-to graph for each program point. In the absence of recursion, the analysis processes each advice or method once to obtain a general analysis result that can be specialized for each calling context. In the presence of recursion it uses a fixed point algorithm within each strongly connected component of the call graph.

The analysis distinguishes several kinds of nodes in the points-to graph. As in the classic object allocation site model [6], there is one *inside* node for each allocation site in the program; this node represents all objects allocated at that site. In addition, the analysis uses two new kinds of nodes: parameter nodes and load nodes. *Parameter* nodes represent objects passed as parameters into the analyzed method. An object *escapes* if it is reachable from outside the currently analyzed method (i.e., it is reachable from the parameters, a static class variable, or an object passed to a parallel thread). *Load* nodes represent objects whose references are read from fields of escaped objects. If an object does not escape it is *captured.*

Parameter and load nodes are essential for achieving a compositional, context-sensitive analysis. The analysis computes one parameterized result (a points-to graph) for the end of each method, and later instantiates this result for the calling context at each call site that may invoke that method.[3] More specifically, at each call site, the interprocedural analysis maps the parameter and load nodes from the invoked method to the corresponding nodes from the calling context. For example, the analysis maps a parameter node to all nodes that the corresponding actual argument points to.

For each method the *analysis domain* consists of the currently analyzed method and all of the methods that it may (transitively) invoke. *Inside* edges represent references created within the analysis domain; *outside* edges represent references read within the domain from escaped objects. The tuple $\langle n_1, f, n_2\rangle$ denotes an edge from n_1 to n_2 along field f.

Figure 10 presents the points-to graph from the end of `Stack.push(int)` from Figure 1. Solid circles represent inside nodes; dashed circles represent parameter and load nodes. Solid arrows represent inside edges; dashed lines represent outside edges. The parameter `this` points to the parameter node P1 that represents the `this` object. `Stack.push(int)` reads the field `this.head`. The node P1 escapes, because it is reachable from the caller. During the analysis of this method, the algorithm does not know what other parts of the program may write into `this.head`. The analysis therefore uses the load node L1 to represent the object read from that field. Next, `Stack.push(int)` allocates a new `Cell` object (represented by the inside node I1), makes it point to `this.head` (the inside edge $\langle$`I1`, `next`, `L1`$\rangle$), and sets `this.head` to point to I1 (the inside edge $\langle$`P1`, `head`, `I1`$\rangle$).

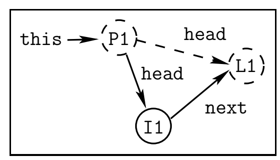

Figure 10: Points-to graph for `Stack.push(int)`.

Intraprocedural analysis: At the start of each method, each object parameter (i.e., not an `int`, `boolean`, ...) p_i points to the corresponding parameter node. Next, our analysis propagates information along the control flow edges, using transfer functions that abstractly interpret [9] statements from the analyzed program. At control flow join points, the analysis merges the incoming points-to graphs: e.g., the resulting points-to graph contains any edge that exists in one or more of the incoming points-to graphs. The analysis iterates over loops until it reaches a fixed point.

Figure 11 presents a graphical representation of several intraprocedural transfer functions (see [24] for a full description). As a general rule, we perform strong updates on variables, i.e., assigning something to a variable removes its previous values, and weak updates on node fields, i.e., the analysis of a store statement that creates a new edge from $n_1.f$ leaves the previous edges in place. Because n_1 may represent multiple objects, all of these edges may be required to correctly represent all of the references that may exist in the heap.

A copy statement "$v_1 = v_2$" makes v_1 point to all the nodes to which v_2 points. A new statement "v = `new` C"

[3]In this section, we use the term "calling context" to denote the points-to graph (i.e., the abstract heap state) at a specific program point.

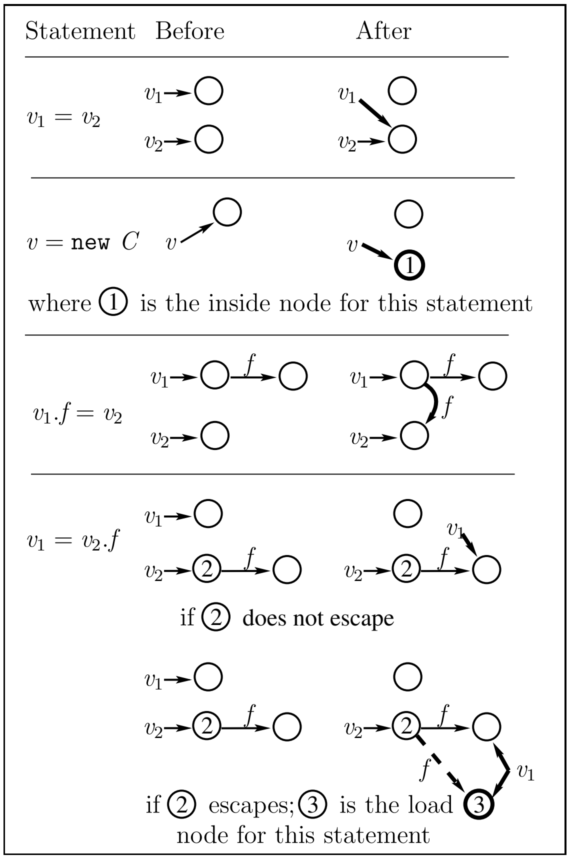

Figure 11: Graphical representation of intraprocedural transfer functions. Solid circles represent inside nodes, dashed circles represent load and parameter nodes, solid arrows represent inside edges, and dashed arrows represent outside edges. Bold circles and lines indicate potentially new nodes and edges.

makes v point to the inside node attached to the label lb. For a store statement "$v_1.f = v_2$", the analysis introduces an f-labeled inside edge from each node to which v_1 points to each node to which v_2 points. The case of a load statement "$v_1 = v_2.f$" is more complex. First, after the load, v_1 points to all the nodes that were pointed by an inside edge from $v_2.f$. If one of the nodes that v_2 points to, say n_2, escapes, a parallel thread or an unanalyzed method may create new edges from $n_2.f$, edges that point to objects created outside the analysis domain. The analysis therefore uses the load node n^L_{lb} for label lb to represent these objects, sets v_1 to point to n^L_{lb} too, and introduces an outside edge from n_2 to n^L_{lb}. The interprocedural analysis uses this outside edge to find nodes from the calling context that may have been loaded at label lb.

Interprocedural analysis: For each call statement "$v_R = v_0.s(v_1, \ldots, v_j)$", the analysis uses the points-to graph G before the call and the points-to graph G_{callee} from the end of the invoked method *callee* to compute a points-to graph for the program point after the call. If there are multiple possible callees (this may happen because of dynamic dispatch), the analysis considers all of them and merges the resulting set of points-to graphs.

The interprocedural analysis operates in two steps. First, the analysis computes a node mapping that maps the parameter and load nodes from *callee* to the nodes they represent in the caller. Initially, the analysis maps each parameter node to the nodes to which the corresponding actual argument points. It then uses two rules to match outside edges (from read operations) against inside edges (from corresponding write operations) and discover additional node mappings. The first rule matches outside edges from the callee against inside edges from the caller. This rule handles the case where the callee reads data from the calling context. If node n_1 maps to node n_2, we map each outside edge $\langle n_1, f, n_3 \rangle$ from G_{callee} against each inside edge $\langle n_2, f, n_4 \rangle$ from G, and add a mapping from n_3 to n_4. The second rule maps outside and inside edges from the callee. This rule resolves any aliasing introduced at the calling context. If nodes n_1 and n_2 have a common mapping, they may represent the same location. Therefore, we match each callee outside edge $\langle n_1, f, n_3 \rangle$ from G_{callee} against each callee inside edge $\langle n_2, f, n_4 \rangle$ and map n_3 to n_4.

Next, the analysis uses the node mapping to project G_{callee} and merge it with the points-to graph from before the call. The full details are presented in [24].

3.3 Effect Analysis

We have updated the pointer analysis from Section 3.2 to maintain a set of read and write effects $E(m)$ for each analyzed method m. Each time the analysis of a method m encounters a load/store instruction, it records into $E(m)$ the relevant field and node(s). For example, the analysis of the `Stack.push(int)` method records, among other effects, a write of the field `head` of the P1 node in Figure 10 that represents the receiver object (`this`).

The analysis propagates effects interprocedurally as follows: when the analysis of a method m encounters a call instruction, it uses the interprocedural node mapping to project the effects of the callee and include these effects in the set $E(m)$. For example, suppose the analysis of m encounters a call to the method `Stack.push(int)` from Figure 1. If P1 maps to the nodes I8 and I9 (for example), the write effect on P1.`head` projects into write effects on I8.`head` and I9.`head`. The analysis adds these two effects to the set $E(m)$ of method m's effects.

The analysis does not record effects to captured nodes since these nodes represent objects that are unreachable and therefore invisible outside the analyzed method.

Once the combined pointer and effect analysis terminates, it generates the scope of each analyzed method by projecting out the recorded field accesses. Note that the analysis works with a representation that maintains more information than the generated scopes. In particular, the escape information enables the analysis to recognize (and eliminate) any accesses to captured nodes. The analysis information also enables it to recognize accesses that occur only on new objects (i.e., objects represented by inside nodes).

For each `proceed` statement, the analysis marks the nodes that `proceed`'s arguments point to as escaped and uses a special *return* node to model the possible result of the `proceed`. Note that the analysis does not treat `proceed` as a call to the original code at the join point: the join point code is not part of the advice. In particular, the effects of the join point code are not included in the effects of the advice.

The analysis uses manually generated points-to graphs and effects for commonly used native methods. It (conservatively) assumes that all other native methods mutate all nodes reachable from their arguments.

3.4 Control Flow Analysis

The control flow analysis processes the control flow graph (CFG) of each advice to classify the advice as augmentation, narrowing, replacement, or combination advice. We discuss only the case of around advice. Before and after advice are special cases of around advice: we can desugar each before advice into an around advice terminated with a `proceed` statement and similarly for each after advice.

The CFG of each advice models both normal and exceptional control flow. Each CFG path starts in a special entry node and ends either in a special normal exit node (for executions that return normally) or in a special exceptional exit node (for executions that terminate due to uncaught exceptions). Each `throw` statement generates control flow edges to the appropriate handlers; if the handlers are not guaranteed to catch the exception, we also have a control flow edge to the exceptional exit node. We have similar control flow edges for each method call that may throw an uncaught exception. To focus the attention of the developer on the important part of the program, we consider only the explicitly thrown exceptions and not the runtime exceptions introduced by the Java semantics (for example, `NullPointerException` or `ArithmeticException`). Similarly, we assume that library methods do not throw uncaught exceptions.

We can now give a more technical definition of our classification. First, we classify as combination advice 1) any advice with a `proceed` statement with a different list of arguments than the original join point, and 2) any advice that returns a value different from the value returned by `proceed`. We check these conditions using def-use chains. We classify the remaining advice based on the properties of the CFG paths:

- **Augmentation Advice:** An advice is augmentation advice if every path in its CFG contains exactly one `proceed` statement and terminates in the normal exit node.[4] This property ensures that any execution of the advice executes the join point exactly once, plus some additional code from the advice.

- **Replacement Advice:** An advice is replacement advice if no path in its CFG contains a `proceed` statement. This property ensures that the advice completely replaces the execution of the join point.

- **Narrowing Advice:** An advice is narrowing advice if each path in its CFG belongs to one of the following two categories: 1) it does not contain a `proceed` statement, or 2) it contains exactly one `proceed` statement and terminates in the normal exit node. This property ensures that the advice either executes the join point or bypasses it using normal or exceptional control flow.

- **Combination Advice:** All other advice.

Our control flow analysis performs this classification using a series of graph reachability tests on the method control flow graph (CFG). We perform the following tests, in the order below, and we stop as soon as a test succeeds:

[4]Advice that throws an exception after executing the original join point is classified as combination advice.

1. If no `proceed` is reachable from the method entry, then the advice is a replacement advice.

2. If a `proceed` statement is reachable from (possibly another) `proceed` statement, then the advice may execute a `proceed` statement more than once. In this case, the advice is a combination advice.

3. If the exceptional exit node is reachable from any of the `proceed` statements, then the advice is a combination advice.

4. If the normal and exceptional method exit nodes are unreachable in the CFG without the nodes for the `proceed` statements, then any path executes `proceed` exactly once. Hence, the advice is an augmentation advice.

5. Otherwise, the advice is a narrowing advice.

4. REPORT GENERATION

A report generator produces a report for each aspect that summarizes the interactions between each advice in the aspect and the other advice and methods in the program. Figures 7 and 9 present examples of generated reports. For each advice in the aspect, the report generator produces the following information:

- **Advice Classification:** A classification of the advice as augmentation, narrowing, replacement, or combination advice.

- **Interactions:** For each other advice or method, a classification of the interaction between the advice and the other advice or method as orthogonal, independent, observation, actuation, or interference. Crosscut methods appear before methods that the advice does not crosscut.[5]

  The report generator prints a list of all memory conflicts between the advice and the method, in the order of the conflict severity: it reports write/write conflicts, write/read, read/write, and read/read conflicts in this order. For each category, it lists the affected fields.

This report is designed to enable the developer to quickly recognize interactions that support modular reasoning: if the aspect is an augmentation aspect and all of the interactions are orthogonal, independent, or observation, the aspect does not affect the operation of the original program within its original scopes. The developer's understanding of the original program therefore remains completely valid when reasoning about the crosscut program.

If, on the other hand, the aspect may interfere with the execution of one or more of the methods, the report identifies the methods and fields involved in the interference. The developer can then focus on those methods and fields when reasoning about the consequences of crosscutting the program with the aspect.

[5]We conservatively approximate the pointcut designator of the advice. Specifically, we assume that the dynamic part of the pointcut designator always matches. It would be possible to increase the precision by using more sophisticated analyses to more precisely resolve dynamic conditions involving the control flow, parameter values, and values of object fields.

It is possible to configure the report generator to produce different amounts of information. For example, it is possible to configure the report generator to suppress interactions involving specific fields. Such configurations are occasionally useful to help eliminate the explicit presence of well-understood interactions and focus the attention of the developer on any remaining interactions.

By default, the current report generator produces shorter reports that are designed to more efficiently direct the programmer to any potentially problematic interactions. Specifically, the current report generator does not report independent, orthogonal, or observation interactions. It also does not report fields already reported in a more severe conflict category (for example, it does not report a write/read conflict for a field that already has a write/write conflict for the same advice-method interaction pair).

The current report generator is designed to help developers trace potential interactions starting from a given aspect. It is possible to reformat the same information to help developers trace interactions in different directions starting from a given method or field. The reported information could also serve as the foundation for an interactive system that would allow the developer to flexibly trace potential interaction relationships through aspects, methods, and fields.

5. EXPERIENCE

We implemented our system using a combination of the AspectJ compiler [11] and the MIT Flex compiler infrastructure [1]. Specifically, we use the AspectJ compiler to generate bytecodes for the aspects, which enables our analysis (which is implemented in the MIT Flex compiler infrastructure) to analyze them. To gain experience with our implemented system, we obtained several aspect-oriented programs and used our system to analyze them.

5.1 Basic Aspects

Many aspects are designed to add tracing, monitoring, logging, or consistency checking functionality to crosscut classes [17, 16]. To test the functionality of our implementation and evaluate its ability to verify the properties that we expected the aspect to have, we analyzed three simple programs (in addition to the two examples from Section 2.)

We developed a simple `NullChecker` aspect that uses before advice to check that certain methods are called with non-null arguments only. Our analysis classified this aspect as a narrowing aspect with Orthogonal interactions.

The exception logging aspect from Section 5.4.2 of [17] contains after advice that logs any exception that is thrown from a method invocation. Our analysis recognized this advice as augmention advice that writes the abstract field that represents the input/output state. This advice has Actuation or Interference interactions with advice or methods that perform input or output operations. As mentioned in Section 4, it is possible to configure the report generator to omit these interactions once the developer understands them.

The tracing aspect from Section 2.4.2 of [17] contains before advice and after advice that print short messages before and after each join point. The two pieces of advice use the aspect field `_callDepth` to maintain an appropriate indentation length. Our analysis recognized both pieces of advice as augmentation advice; each advice has an Interference interaction (because of the write to `_callDepth`) with the other advice and with the constructor of the aspect. In addition, each advice has an Actuation interaction with the method that does the actual printing because this method reads `_callDepth`. This printing method is invoked only from within the tracing aspect and is not part of the core program.

5.2 Telecom

The core Telecom program (available at `www.aspectj.org`) simulates a community of phone users. The `Timing` aspect adds functionality to record the phone connection time; the `Billing` aspect uses the connection time to bill the originator of the call.

The reports show that although the `Timing` and `Billing` aspects read fields that the Telecom program writes, they do not write any of these fields. They therefore have at most Observation interactions with the methods in the Telecom program, leaving its behavior intact within its original scopes. There is, however, an Actuation interaction between the `Timing` and `Billing` advice — the `Timing` advice writes a field that the `Billing` advice reads. Further investigation reveals that this interaction is anticipated and desirable because this field carries information about the connection time from the `Timing` aspect to the `Billing` aspect.

5.3 Aspects for Business Rule Implementation

The example from Section 12.5 of [17] uses aspects to implement business rules in a banking system. `MinimumBalanceRuleAspect` has an advice that crosscuts each debit operation and throws an exception if the operation would decrease the account balance below a certain threshold value. Our analysis identified this advice as narrowing advice that has at most Observation interactions with methods in the original banking system. `OverdraftProtectionRuleAspect` has an advice that crosscuts each check clearance operation and, if necessary, transfers money from an overdraft account into the checking account before executing the check clearance. An appropriate exception is thrown if the overdraft account has insufficient funds. The analysis identified this advice as narrowing advice that has an Actuation or Interference interaction with methods that may read or write the account balance field. The analysis also detected that the other aspects from the example (three logging examples) contain only augmentation advice. These pieces of advice have Interference or Actuation interactions with methods and advice that contain input or output operations and at most Observation interactions with the remaining methods in the core program.

5.4 Spacewar

We also obtained the Spacewar program from `www.aspectj.org`. This program comes with several synchronization aspects that allow the programmer to declare groups of mutually exclusive methods. The aspects then crosscut the program to add the synchronization required to enforce the mutual exclusion. The `GameSynchronization` and `RegistrySynchronization` aspects apply this basic strategy to two different groups of methods. Both of these aspects use encapsulated `Vector`s and `Hashtable`s to record groups of mutually exclusive methods. In our original analysis, these encapsulated data structures generated false Interference interactions with other aspects and methods of the program that access other `Vector`s or `Hashtable`s. We therefore extended our analysis to automatically recognize encapsulated objects (in some cases using information about the standard Java collection classes). This extension enabled the analysis to lift reads and writes on encapsulated objects into corresponding reads and writes on the encapsulating object. After this extension, the generated reports indicate that the synchronization advice has only Orthogonal or Observation interactions with all of the methods in the original program, providing the developer with a guarantee that the aspects add new functionality that does not interfere with the existing Spacewar functionality.

Our system also successfully analyzed the other aspects. The `EnsureShipIsAlive` aspect contains narrowing advice that prevents console commands from operating on destroyed ships. The `RegistrationProtection` aspect contains replacement advice that prevents calls to the registration subsystem from taking place anywhere except from within the methods that construct or terminate the registered object.

The remaining three aspects in the program contain a total of seven pieces of augmentation advice. These pieces of advice are responsible for the look and feel of the program: they set up the display and (re)paint the game objects. There are Observation, Actuation, and Interference interactions between these pieces of advice and the `java.awt` library methods that maintain the graphical state.

Spacewar is our largest program, with approximately 2000 lines of Java code in the core program (Telecom is second largest, with approximately 700 lines of Java code in the core). In addition to this code, our analysis processes a significant fraction of the Java standard libraries.

5.5 Discussion

In general, we found that our analysis was quite effective at verifying the lack of interference between the core program and tracing and logging aspects. For layered aspects such as our suite of Telecom aspects, our system was able to verify the lack of interference between different layered pieces of advice (in addition to verifying the lack of interference between the advice and methods in the core program).

Our experience with the banking system aspects (see Section 5.3) suggests that it might be worthwhile to incorporate a purity analysis into our system. This extension would provide another useful concept for developers — Pure advice would have no externally visible side effects at all. It would therefore have only Orthogonal, Independent, or Observation interactions with other advice or methods and exist only for its effect on the flow of control or the call/return interface. Our current pointer and effect analysis can already check advice or method purity [25].

In general, the aspect-oriented programs that we were able to obtain were smaller than the more traditional Java programs that we are used to working with. One can view this fact as either evidence of the ability of aspects to deliver a lot of functionality with little code or as an inevitable result of the relatively recent widespread availability of aspect-oriented languages. In any case, our past results (and those of others) indicate that our analysis should scale to larger programs [29, 5, 7, 4, 24]. Like all automated analysis tools, we expect the value of our tool to increase in proportion to the complexity of the programs on which it is deployed.

Some traditional reasoning approaches rely on tracing explicit control flow paths through the program. The implicit control flow paths in aspect-oriented programs complicate this kind of reasoning — they introduce the possibility that the code that the developer sees when he or she traces an explicit control flow path may not be all (or even any) of the code that will actually execute when the program runs. One appropriate response to this situation is to embrace, when justified, the new modularity mechanisms that aspect-oriented programming provides and to use systems such as ours to help developers reason effectively about the resulting aspect-oriented programs. Our experience supports this point of view: our analysis and classification system helped us to understand and appreciate the functionality decomposition and resulting modularity properties of our programs.

6. RELATED WORK

Researchers have explored a variety of language features and mechanisms that allow the developer to separate the code for different concerns into separate syntactic units, then compose these units (either statically or dynamically) to obtain the final behavior [20, 15, 14, 13, 21, 22, 19, 3, 2]. The primary goal is to provide the mechanisms required to modularize a program with crosscutting concerns. Our research assumes that these kinds of mechanisms are useful and will become widely used in practice; our contribution is a classification system and analysis that developers can use to reason about the resulting crosscut programs. Although we have designed our system to work with AspectJ, we believe the basic principles and analysis ideas will apply whenever the language supports crosscutting concerns that may access shared object fields or affect the control flow.

We believe that most aspect-oriented programmers have an intuitive feel for aspects that enable modular reasoning and make value judgements that distinguish these aspects from more invasive aspects [12, 17]. Based on their experience using aspect-oriented programming, researchers have proposed design rules that limit the potential interactions between the aspect and the crosscut method [12]. The proposed control flow restrictions are essentially equivalent to requiring the aspect to be an augmentation aspect.

Researchers have proposed the isolation of crosscutting concerns to support the modular verification of temporal properties of state machines [18, 10]. Our research differs in its focus on standard programming languages and in its emphasis on indirect interactions (both intended and unexpected) that occur because of accesses to object fields.

Developers can use aspect-oriented programming to add new methods to classes and to modify the class hierarchy. Composition systems such as Hyper/J [28] allow developers to compose class hierarchies. Both approaches can change the explicit control flow in the program (because they may change the methods invoked at dynamic dispatch sites). Researchers have developed analyses to identify such potential changes [27, 26]. These analyses are orthogonal to ours: our analysis focuses on implicit control-flow interactions that take place when an advice crosscuts a method and on indirect interactions that take place when advice and methods access the same object fields.

The research most closely related to ours classifies aspects into *spectators* and *assistants* [8]. A spectator does not affect the control flow into or out of a crosscut method and writes only private data that is unavailable to crosscut methods. Spectators correspond to augmentation aspects in our classification system with the additional requirement that they write private data only. This restriction ensures that they have only orthogonal, independent, or observation interactions with crosscut methods. In this classification system, all other aspects are assistants. Because assistants can change the behavior of the crosscut module, the proposal is that the crosscut module must explicitly accept the assistance by referencing the crosscutting aspect by name.

Both this research and our research identify a set of precise criteria that classify aspects into categories and relate these categories to the ability of the developer to reason about the resulting crosscut program. Our research differs in that it has a more sophisticated classification scheme and an implemented analysis that automatically applies the classification scheme. In addition, we accept the possibility that some legitimate aspect-oriented programs may not fit into any of the categories in our system that are known to support modular reasoning. We have therefore designed our classification scheme and analysis reports to provide guidance that helps the developer reason effectively about the interactions in these programs.

7. CONCLUSION

Aspect-oriented languages support new and powerful program structuring techniques that promise significant modularity benefits if used appropriately. But if used unwisely, aspect-oriented constructs can interfere with the abstractions in the program and substantially complicate the ability of the developer to reason about its behavior.

This paper makes the case for attacking this problem with a classification scheme backed by a program analysis that automatically classifies aspects and interactions between aspects and methods. This combination enables the developer to quickly recognize interaction patterns that support modular reasoning and to focus in on the causes of potentially nonmodular interactions. We believe the eventual result will prove to be a significant improvement in our ability to develop and reason about aspect-oriented programs, with a corresponding increase in the delivered utility of this promising program structuring technique.

Acknowledgments

The authors would like to acknowledge the participants of the New Directions in Software Technology (NDIST03) workshop for the interactions that inspired this research. The authors would also like to thank Patrick Lam for his help with the Soot compiler infrastructure [23] and C. Scott Ananian for his help with the Flex compiler infrastructure [1].

8. REFERENCES

[1] C. S. Ananian. MIT FLEX compiler infrastructure for Java. Available from http://www.flex-compiler.lcs.mit.edu.

[2] D. Batory and S. O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, 1(4), Oct. 1992.

[3] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the ACM*, Oct. 2001.

[4] B. Blanchet. Escape analysis for object oriented languages. Application to Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[5] J. Bogda and U. Hoelzle. Removing unnecessary synchronization in java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[6] D. Chase, M. Wegman, and F. Zadek. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Program Language Design and Implementation*, pages 296–310, White Plains, NY, June 1990. ACM, New York.

[7] J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[8] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. Technical Report TR 02-04, Department of Computer Science, Iowa State University, Mar. 2002.

[9] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th POPL*, 1977.

[10] K. Fisler and S. Krishnamurthi. Modular verification of collaboration-based software designs. In *Proceedings of the Joint European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Vienna, Austria, Sept. 2001.

[11] G. Kiczales etal. AspectJ compiler. Available from http://eclipse.org/aspectj.

[12] M. Kersten and G. Murphy. Atlas: a case study in building a web-based learning environment using aspect-oriented programming. In *Proceedings of the 1999 Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. Getting started with AspectJ. *Communications of the ACM*, 15(12), Oct. 2001.

[14] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An overview of AspectJ. In *Proceedings of the 2001 European Conference on Object-Oriented Programming*, Budapest, Hungary, June 2001.

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of the 1997 European Conference on Object-Oriented Programming*, Jyvaskyla, Finland, June 1997.

[16] I. Kiselev. *Aspect-Oriented Programming with AspectJ*. Sams Publishing, 2002.

[17] R. Laddad. *ApectJ in Action*. Manning Publications Company, Greenwich, CT, 2003.

[18] H. Li, S. Krishnamurthi, and K. Fisler. *Verifying Cross-Cutting Features as Open Systems*. Manning Publications Company, Greenwich, CT, 2003.

[19] K. Lieberherr, D. Lorenz, and M. Mezini. Programming with aspectual components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Mar. 1999.

[20] D. Moon. Object-oriented programming with flavors. In *Proceedings of the 1986 Conference on Object-Oriented Programming Systems, Languages and Applications*, Portland, OR, Nov. 1986.

[21] H. Ossher and P. Tarr. Multi-dimensional separation of concerns in hyperspace. Technical Report RC 21452(96717)16APR99, IBM T.J. Watson Research Center, 1999.

[22] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of the ACM*, Oct. 2001.

[23] Sable compiler group at McGill. Soot: a Java optimization framework. Available from http://www.sable.mcgill.ca/soot.

[24] A. Salcianu. Pointer analysis and its applications to Java programs. Master's thesis, MIT Laboratory for Computer Science, 2001.

[25] A. Salcianu and M. Rinard. A combined pointer and purity analysis for Java programs. Technical Report MIT-CSAIL-TR-949, MIT CSAIL, 2004.

[26] G. Snelting and F. Tip. Semantics-based composition of class hierarchies. In *Proceedings of the 2002 European Conference on Object-Oriented Programming*, 2002.

[27] M. Stoerzer and J. Krinke. Interference analysis for AspectJ. In *Workshop on Foundations of Aspect-Oriented Languages*, 2003.

[28] P. L. Tarr, H. Ossher, W. H. Harrison, and S. M. S. Jr. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, 1999.

[29] J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th Annual Conference on Object-Oriented Programming Systems, Languages and Applications*, Denver, CO, Nov. 1999.