Translucid Contracts for Aspect-oriented Interfaces

Mehdi Bagherzadeh
Iowa State University,
Ames, IA, USA
mbagherz@cs.iastate.edu

Hridesh Rajan
Iowa State University,
Ames, IA, USA
hridesh@cs.iastate.edu

Gary T. Leavens
University of Central Florida,
Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

There is some consensus in the aspect-oriented community that a notion of interface between joinpoints and advice may be necessary for improved modularity of aspect-oriented programs, for modular reasoning, and for overcoming pointcut fragility. Different approaches for adding such interfaces, such as aspect-aware interfaces, pointcut interfaces, crosscutting interfaces, explicit joinpoints, quantified typed events, open modules, and joinpoint types decouple aspects and base code, enhancing modularity. However, existing work has not shown how one can write specifications for such interfaces that will actually allow modular reasoning when aspects and base code evolve independently, and that are capable of specifying control effects, such as when advice does not proceed. The main contribution of this work is a specification technique that allows programmers to write modular specification of such interfaces and that allows one to understand such control effects. We show that such specifications allow typical interaction patterns, and interesting control effects to be understood and enforced. We illustrate our techniques via an extension of Ptolemy, but we also show that our ideas can be applied in a straightforward manner to other notions of joinpoint interfaces, e.g. the crosscutting interfaces.

1. INTRODUCTION

In the past decade, the remarkable visibility and adoption of aspect-orientation [28] in research and industrial settings only confirms our belief that new AOSD techniques provide software engineers with valuable opportunities to separate conceptual concerns in software system to enable their independent development and evolution. This same decade of AOSD research has also witnessed an intense debate surrounding two issues: pointcut fragility and modular reasoning. The debate on pointcut fragility focuses on the use of pattern matching as a quantification mechanism [48, 50], whereas that on modular reasoning focuses on the effect of AO modularization on independent understandability and analyzability of modularized concerns [1, 17, 26, 29]. Although the jury is still out, in the later part of the last decade some consensus has begun to emerge that a notion of interfaces may help address questions of pointcut fragility and modular reasoning [1, 13, 20, 29, 41, 47, 49].

1.1 The Problems and their Importance

Although these proposals differ significantly in their syntactic forms and underlying philosophies, the permeating theme is that they provide some notion of explicit interface that abstracts away the details of the modules that are advised (typically referred to as the "base modules") thus hiding such details from modules that advise them (typically referred to as the "crosscutting modules" or "aspects"). Leaving the comparison and contrast of software engineering properties of these proposals to empirical experts, in this paper we focus on studying the effectiveness of such interfaces towards enabling a design by contract methodology for AOSD [1].

Design by contract methodologies for AOSD have been explored before [49, 55], however, existing work relies on behavioral contracts. Such behavioral contracts specify, for each of the aspect's advice methods, the relationships between its inputs and outputs, and treat the implementation of the aspect as a black box, hiding all the aspect's internal states from base modules and from other aspects. To illustrate, consider the snippets shown in Figure 1 from the canonical drawing editor example with functionality to draw points, lines, and a display updating functionality.

Figure 1 uses a proposal for aspect interfaces[2], promoted by our previous work on the Ptolemy language [41]. In Ptolemy, programmers declare event types that are abstractions over concrete events in the program. Lines 10–16 declare an event type that is an abstraction over program events that cause change in a figure. An event type declaration may declare variables that make some context available. For example, on line 11, the changing figure, named `fe`, is made available. Concrete events of this type are created using **`announce`** expressions as shown on lines 5–7.

A significant advantage of such interfaces is that they provide a syntactic location to specify contracts between the aspect and the base code [49] that is independent of both. Following previous work [49, 55], we have added an example behavioral contract to the interface (event type `Changed`) (lines 12-15). This behavioral contract is written in a form similar to our proposal to make comparisons easier. This contract states that any concrete event announcement must ensure that the context variable `fe` is non null and observers (e.g. the `Update` class on lines 17–26) for this event must not modify `fe`.

The first problem with specifying aspect interfaces using behavioral contracts is that they are insufficient to specify the control effects of advice in full generality. For example, with just the be-

[1]This is not to be confused with DBC *using* AOP, where the advice construct is used to represent contract of a method. Rather we speak of the contract between aspects and the base code.

[2]The choice is more of preference than of necessity. Other proposals are equally suitable for this discussion. The reader is encouraged to consider alternatives discussed in Section 4.

```
1 class Fig { }
2 class Point extends Fig {
3  int x; int y;
4  Fig setX(int x){
5   announce Changed(this){
6    this.x = x; this
7   }
8  }
9 }
```

```
10 Fig event Changed {
11  Fig fe;
12  provides  fe != null
13  requires {
14    fe == old(fe)
15  }
16 }
```

```
17 class Update {
18  Update init(){register(this)}
19  when Changed do update;
20  Display d;
21  Fig update(thunk Fig rest,
22                   Fig fe){
23   d.update(fe);
24   invoke(rest)
25  }
26 }
```

Figure 1: A behavioral contract for aspect interfaces using Ptolemy [41] as the implementation language. See Section 2.1 for syntax.

havioral specification of the event type `Changed`, we cannot determine whether the body of the method `setX` will always set the current `x` coordinate to the argument. Such assertions are important for reasoning, which depends on understanding the effect of composing the aspect modules with the base code [44, 49]. In Figure 1, for example, the behavioral contract for `Changed` doesn't serve to alert us to an (inadvertently) missing **`invoke`** expression from the `Update` code that would skip the evaluation of the expression **`this`**`.x = x` in the method `setX`. In AspectJ terms this would be equivalent to a missing **`proceed`** statement from an **`around`** advice. Ideas from Zhao and Rinard's Pipa language [55], if applied to AO interfaces help to some extent, however, as we discuss in greater detail in Section 5, Pipa's expressiveness beyond simple control flow properties is limited.

The second problem with such behavioral contracts is that they don't help us in effectively reasoning about the effects of aspects on each other. Consider another example concern, say `Logging`, which logs the event `Changed`. For this concern different orders of composition with the `Update` concern in Figure 1 could lead to different results. In Ptolemy ordering between aspects can be specified using **`register`** expressions that activate an aspect. In AspectJ **`declare precedence`** serves the same purpose. In one composition where `Update` runs first followed by `Logging`, the evaluation of `Logging` will be skipped, whereas `Logging` would work in the reverse order of composing these concerns. An aspect developer may not, by just looking at the behavioral contract of the aspect interface, reason about their aspect modules. Rather they must be aware of the effects of all aspects that apply to that aspect interface [1, 16, 17]. Furthermore, if any of these aspect modules changes (i.e., if their effects change), one must reason about every other aspect that applies to the same aspect interface.

Finally, even if programmers don't use formal techniques to reason about their programs, contracts for AO interfaces can serve as the programming guidelines for imposing design rules [49,52]. Behavioral contracts for AO interfaces yield insufficiently specified design rules that leave too much room for interpretation, which may differ significantly from programmer to programmer. This may cause inadvertent inconsistencies in AO program designs and implementations, leading to hard to find errors.

1.2 Contributions to the State-of-the-art

The main contribution of this work is the notion of *translucid contracts* for AO interfaces, which is based on grey box specifications [9–12]. A translucid contract for an AO interface can be thought of as an abstract algorithm describing the behavior of aspects that apply to that AO interface. The algorithm is abstract in the sense that it may suppress many actual implementation details, only specifying their effects using specification statements. This allows the specifier to decide to hide some details, while revealing others. As in the refinement calculus, code satisfies an abstract algorithm specification if the code refines the specification [2,36,37], but we use a restricted form that requires structural similarity, to allow specification of control effects.

To illustrate, consider the translucid contract shown in Figure 2. The classes `Fig` and `Point` in this example are the same as in Figure 1. Contrary to the behavioral contract, internal states of the handler methods that run when the event `Changed` is announced are exposed. In particular, any occurrence of **`invoke`** expression in the handler method must be made explicit in the translucid contract. This in turn allows the developer of the class `Point` that announces the event `Changed` to understand the control effects of the handler methods by just inspecting the specification of `Changed`. For example, from lines 5–6 one may conclude that, irrespective of the concrete handler method, the body for the method `setX` on line 6 of Figure 1 will always be run. Such conclusions allow the client of the `setX` to make more expressive assertions about its control flow without considering every handler method that may potentially run when the event `Changed` is announced.

```
1 Fig event Changed {
2  Fig fe;
3  provides  fe != null
4  requires {
5   preserves fe == old(fe);
6   invoke(next)
7  }
8 }
9 class Update {
10  /* ... the same as before */
11  Fig update(thunk Fig rest, Fig fe){
12   refining preserves fe==old(fe){
13    d.update(fe);
14   } ;
15   invoke(rest)
16  }
17 }
```

Figure 2: A translucid contract for aspect interfaces

Making the **`invoke`** expression explicit also benefits other handlers that may run when the event `Changed` is announced. For example, consider the logging concern discussed earlier. Since the contract of `Changed` describes the control flow effects of the handlers, reasoning about the composition of the handler method for logging and other handler methods becomes possible without knowing about all explicit handler methods that may run when the event `Changed` is announced. In this paper we explicitly focus on the use of translucid contract for describing and understanding control flow effects.

To soundly reap these benefits, the translucid contract for the event type `Changed` must be refined by each conforming handler method [2, 36, 37]. We borrow the idea of structural refinement from JML's model programs [45] and enhance it to support aspect-oriented interfaces, which requires several adaptation that we discuss in the next section. Briefly the handler method `update` on lines 11–16 in Figure 2 refines the contract on lines 5–6 because line 15 matches line 6 and lines 12–14 claim to refine the spec-

ification expression on line 5. In summary, this work makes the following contributions. It presents:

- A specification technique for writing contracts for AO interfaces;
- An analysis of the effectiveness of our contracts using Rinard *et al.*'s work [44] on aspect classification that shows that our technique works well for specifying all classes of aspects (as well as others that Rinard *et al.* do not classify);
- A demonstration that besides the AO interface proposal by the previous work of Rajan and Leavens [41], our technique works quite well for crosscut interfaces [49] and Aldrich's Open Modules [1] and a discussion of the applicability of our technique to Steimann *et al.*'s joinpoint types [47], Hoffman and Eugster's explicit joinpoints [21], and Kiczales and Mezini's aspect-aware interface [29]; and
- A comparison and contrast of our specification and verification approach with related ideas for AO contracts.

2. TRANSLUCID CONTRACTS

In this section, we describe our notion of translucid contracts and present a syntax and refinement rules for checking these contracts. We use our previous work on the Ptolemy language [41] for this discussion.[3] However, as we show in Section 4 our basic ideas are applicable to other aspect-oriented programming models. We first present Ptolemy's programming features and then describe the specification features.

2.1 Program Syntax

Ptolemy is an object-oriented (OO) programming language with support for declaring, announcing, and registering with events much like implicit-invocation (II) languages such as Rapide [34]. The registration in Ptolemy is, however, much more powerful compared to II languages as it allows developers to quantify over all subjects that announce an event without actually naming them. This is similar to aspect-oriented languages such as AspectJ [27]. The formally defined OO subset of Ptolemy shares much in common with MiniMAO$_1$ [15], a variant of Featherweight Java [22] and Classic Java [18]. It has classes, objects, inheritance, and subtyping, but it does not have **super**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods.

The syntax of Ptolemy executable programs is shown in Figure 3 and explained below. A Ptolemy program consists of zero or more declarations, and a "main" expression (see Figure 1). Declarations are either class declarations or event type declarations.

Declarations. We do not allow nesting of *decl*s. Each class has a name (c) and names its superclass (d), and may declare finite number of fields (*field**) and methods (*meth**). Field declarations are written with a class name, giving the field's type, followed by a field name. Methods also have a C++ or Java-like syntax, although their body is an expression. A binding declaration associates a set of events, described by an event type (p), to a method [41]. An example is shown in Figure 1, which contains a binding on line 19. This binding declaration tells Ptolemy to run the method `update` whenever events of type `Changed` are executed. Implicit invocation terminology calls such methods *handler methods*.

An event type (**event**) declaration has a return type (c), a name (p), zero or more context variable declarations (*form**), and a translucid contract (*contract*). These context declarations specify the types and names of reflective information exposed by conforming events [41]. An example is given in Figure 1 on lines 10–16. In writing examples of event types, as in Figure 1, we show each formal parameter declaration (*form*) as terminated by a semicolon (**;**). In examples showing the declarations of methods and bindings, we use commas to separate each *form*.

[3] Our descriptions of Ptolemy's syntax and semantics are adapted from our previous work [41].

```
prog    ::= decl* e
decl    ::= class c extends d { field*meth* binding* }
          | t event p { form* contract }
field   ::= t f;
meth    ::= t m (form*) { e } | t m (thunk t var, form*) { e }
form    ::= t var,    where var≠this
binding ::= when p do m
e       ::= n | var | null | new c() | e.m(e*) | e.f | e.f = e
          | if (ep) { e } else { e } | cast c e | form = e; e | e; e
          | while (ep) { e } | register(e) | invoke(e)
          | announce p (e*) { e } | refining spec { e }
ep      ::= n | var | ep.f | ep != null | ep == n | ep < n | ! ep | ep && ep
```

where

n	$\in$	$\mathcal{N}$, the set of numeric, integer literals
c, d	$\in$	$\mathcal{C}$, a set of class names
t	$\in$	$\mathcal{C} \cup \{\textbf{int}\}$, a set of types
p	$\in$	$\mathcal{P}$, a set of event type names
f	$\in$	$\mathcal{F}$, a set of field names
m	$\in$	$\mathcal{M}$, a set of method names
var	$\in$	$\{\textbf{this}\} \cup \mathcal{V}$, $\mathcal{V}$ is a set of variable names

Figure 3: Ptolemy's Syntax [41]. Note the new expression `refining` and contracts in the syntax of event types.

Expressions. The formal definition of Ptolemy is given as an expression language [41]. It includes several standard object-oriented (OO) expressions and also some expressions that are specific to announcing events and registering handlers. The standard OO expressions include object construction (**new** c `()`), variable dereference (*var*, including **this**), field dereference ($e.f$), **null**, cast (**cast** t e), assignment to a field ($e_1.f = e_2$), a definition block (t *var* = e_1 **;** e_2), and sequencing (e_1 **;** e_2). Their semantics and typing is fairly standard [13, 15, 41] and we encourage the reader to consult [41].

There are also three expressions pertinent to events: **register**, **announce**, and **invoke**. The expression **register**(e) evaluates e to an object o, registers o by putting it into the list of active objects, and returns o. The list of active objects is used in the semantics to track registered objects. Only objects in this list are capable of advising events. For example line 18 of Figure 1 is a method that, when called, will register the method's receiver (**this**). The expression **announce** p $(v_1, \ldots, v_n)$ $\{e\}$ declares the expression e as an event of type p and runs any handler methods of registered objects (i.e., those in the list of active objects) that are applicable to p [41]. The expression **invoke**(e) is similar to AspectJ's **proceed**. It evaluates e, which must denote an event closure, and runs that event closure. This results in running the first handler method in the chain of applicable handlers in the event closure. If there are no remaining handler methods, it runs the original expression from the event. **thunk** t ensures that the value of e is an event closure with t being the return type of event closure and hence the type returned by **invoke**(e).

When called from an event, or from **invoke**, each handler method is called with a registered object as its receiver. The call passes an event closure as the first actual argument to the handler method (named `rest` in Figure 1 line 21). Event closures may not be explicitly constructed in programs, neither can they be stored in

fields. They are only constructed by the semantics and passed to the handler methods.

There is one new program expression: refining. A **refining** expression, of the form **refining** *spec* { e }, is used to implement Ptolemy's translucid contracts (see below). It executes the expression e, which is supposed to satisfy the contract *spec*.

2.2 Specification Features

The syntax for writing an event type's contract in Ptolemy is shown in Figure 4. In this figure, all nonterminals that are used but not defined are the same as in Figure 3.

```
contract ::= provides sp requires { se }
spec     ::= requires sp ensures sp
sp       ::= n | var | sp.f | sp != null | sp == n
             | sp == old(sp) | ! sp | sp && sp
             | sp < n

se ::= sp | null | new c() | se.m( se* ) | se.f | se.f = se
       | if (sp) { se } else { se } | cast c se | form = se; se
       | while (sp) { se } | se; se | form = se; se | se; se
       | register( se ) | invoke( se ) | announce p ( e* ) { e }
       | next | spec | either { se } or { se }
```

Figure 4: Syntax for writing translucid contracts

A *contract* is of the form **provides** sp **requires** { se }. Here, sp is a specification predicate as defined in Figure 4 and the body of the contract se is an expression that allows some extra specification-only constructs (such as choice expressions).

As discussed previously, sp is the precondition for event announcement. The specification expression se is the abstract algorithm describing conforming handler methods. If a method runs when an event of type p is announced, then its implementation must refine the contract se of the event type p. For example, in Figure 2 the method `update` (lines 11–16) must refine the contract of the event type `Changed` (lines 5–6).

There are four new expression forms that only appear in contracts: specification expressions, **next** expressions, abstract invoke expressions and choice expressions. A specification expression (*spec*) hides and thus abstracts from a piece of code in a conforming implementation [43,45]. The most general form of specification expression is **requires** sp_1 **ensures** sp_2, where sp_1 is a precondition expression and sp_2 is a postcondition. Such a specification expression hides program details by specifying that a correct implementation contains a **refining** expression whose body expression, when started in a state that satisfies sp_1, will terminate in a state that satisfies sp_2 [43, 45]. In examples we use two sugared forms of specification expression. The expression **preserve** sp is sugar for **requires** sp **ensures** sp and **establish** sp is sugar for **requires** `1` **ensures** sp [43]. Ptolemy uses 0 for "false" and non-zero numbers, such as 1, for "true" in conditionals.

The **next** expression, the **invoke** expression and the choice expression (**either** { se } **or** { se }) are place holders in the specification that express the event closure passed to a handler, the call of an event handler using **invoke** and a conditional expression in a conforming handler method. The choice expression hides and thus abstracts from the concrete condition check in the handler method. For a choice expression **either** { se_1 } **or** { se_2 } a conforming handler method may contain an expression e_1 that refines se_1, or an expression e_2 that refines se_2, or an expression **if** (e_0) { e_1 } **else** { e_2 }, where e_0 is a side-effect free expression, e_1 refines se_1, and e_2 refines se_2.

3. ANALYSIS OF EXPRESSIVENESS

To analyze the expressiveness of translucid contracts, in this section we illustrate their application to specify base-aspect interaction patterns discussed by Rinard *et al.* in a previous work [44]. Rinard *et al.* classify base-aspect interaction patterns into: *direct* and *indirect interference*. Direct interference is concerned about control flow interactions whereas indirect interference refers to data flow properties. Direct interference is concerned about calls to **invoke**, which is the Ptolemy's equivalent of AspectJ's **proceed**. Direct interference is further categorized into 4 classes of: augmentation, narrowing, replacement and combination advices. We use the same classification of base-advice interaction for subject-observer interactions. An example, built upon the drawing editor example, is shown for each category of direct interferences.

3.1 Direct Interference: Augmentation

Informally an augmentation handler is allowed to call **invoke** exactly once. Augmentation handler can be an after or before handler. In after augmentation, after the event body the handlers are always executed. The handler `logit` in observer class `Logging` in Figure 5 is an example of an after augmentation. The classes `Point` and `Fig` are the same as in Figure 1. The requirement is "to log the changes when figures are changed". The handler `logit` causes the event body (line 13) to be run first by calling **invoke** and then logs the change in figure.

```
1  Fig event Changed {
2   Fig fe;
3   provides fe != null
4   requires {
5     invoke(next);
6     preserve fe==old(fe)
7   }
8  }
9  class Logging extends Object {
10  when Changed do logit;
11  Log l; /* ... */
12  Fig logit (thunk Fig rest, Fig fe){
13   invoke(rest);
14   refining preserve fe==old(fe) {
15    l.logChange(fe);fe
16   }
17  }
18 }
```

Figure 5: Specifying after augmentation

The interaction between subject and observer is documented explicitly in the event type specification (`Changed`) shown on lines 3-7. Notice that the **invoke** expressions appears exactly once in the event type contract. Thus the base code developers for classes that announce this event can assume that all handlers advising this event would have exactly one call to **invoke** in their implementation and therefore these handlers would be augmentation handlers. Furthermore, **invoke** is called at the beginning of the contract, requiring event handlers to be run after the event body. This imposes another restriction on the implementation of handlers and conveys to the base code developers that not only the handlers are augmentation handler, but also that they will be run after the event body.

Structural similarity is one criterion to be met by handler implementation to refine event specification. In this example structural similarity mandates the handler implementation to have exactly one call to **invoke** at its beginning. This ensures that all handlers advising the event type `Changed` are of type after augmentation.

3.2 Direct Interference: Narrowing

A narrowing handler calls **invoke** at most once, which implies existence of a conditional statement guarding the calls to **invoke**. To illustrate consider the listing in Figure 6, which shows an example of narrowing handler for the drawing editor example. This handler implements an additional requirement for the editor that "some figures are fixed and thus they may not be changed or moved". To implement this additional constraint the field `fixed` is added to the class `Fig`. For fixed figures the value of this field will be 1 and 0 otherwise. The code for the class `Point` is the same as in Figure 1. To implement the constraint the handler `Enforce` skips invoking the base code whenever the figure is fixed (checked by accessing the field `fixed`).

```
1  class Fig extends Object{ int fixed; }
2  Fig event Changed {
3   Fig fe;
4   provides fe != null
5   requires {
6    if(fe.fixed == 0){
7     invoke(next)
8     preserve fe==old(fe)
9    } else {
10    preserve fe==old(fe)
11   }
12  }
13 }
14 class Enforce extends Object {
15  when Changed do check;
16  /* ... */
17  Fig check(thunk Fig rest, Fig fe){
18   if(fe.fixed == 0){
19    invoke(rest)
20   } else {
21    refining preserve fe==old(fe){
22      fe
23    }
24   }
25  }
26 }
```

Figure 6: Specifying narrowing with a translucid contract

The contract for the event type `Changed` documents the possibility of a narrowing handler on lines 5–11. It does not, however, reveal the actual code of the narrowing handler as long as the hidden code refines the specification on line 8 and 10.

All observer's handler of the event type `Changed` must refine its specification. This means that the implementation of such handlers must structurally match the contract on lines 6–11. The implementation of the handler `Enforce` structurally matches the contract thus it structurally refines it. The true block of the **if** expression on line 18–20 refines the true block of the if expression in the specification on lines 6–9 because the empty expression trivially preserves `fe==old(fe)`. The false block of the **if** expression on line 20–24 refines the false block of the **if** expression in the specification on lines 9–11 because lines 21–23 claim to refine the specification.

3.3 Direct Interference: Replacement

A replacement handler omits the execution of the original event body and instead only runs the handler body. In Ptolemy this can be achieved by omitting the **invoke** expression in the handler, equivalent to not calling **proceed** in an around advice in AspectJ.

Figure 7 shows an example of such handler. The example uses several standard sugars such as += and > for ease of presentation. In this example, the method `moveX` causes a point to move along the x-axis by amount `d`. The handler `scaleit` implements the requirement that the "amount of movement should be scaled by a

```
1  class Point extends Fig {
2   int x; int y;
3   Fig moveX(int d){
4    announce Moved(this,d){
5     this.x += d; this
6    }
7   }
8  }
9  Fig event Moved {
10  Point p;
11  int d;
12  provides p!=null &&
13           d>0
14  requires {
15   preserve p!=null &&
16    p.y == old(p.y)
17  }
18 }
19 class Scale extends Object {
20  when Moved do scaleit;
21  int s; /*scale factor*/
22  Fig scaleit(thunk Fig rest,
23      Point p, int d){
24   refining preserve p!=null
25     && p.y == old(p.y) {
26     p.x += s * d ; p
27   }
28  }
29 }
```

Figure 7: Specifying replacement with a translucid contract

scaling factor `s` defined in class `Scale`". Specification of event type `Moved` documents that a replacement handler will be run when this event is announced by omitting the calls to **invoke** in its contract. The specification also documents the invariants maintained by the handlers that may run when the event is announced.

One requirement when writing translucid contracts is to reveal all calls to **invoke** expression. Therefore, if an event type's contract has no **invoke** expression, none of the event type's handlers are allowed to have an **invoke** expression in their implementation. Otherwise the structural similarity criterion of refinement is violated. The handler `scaleit` correctly refines `Moved`'s contract because its body (line 24-27) matches the specification. There are no invoke expressions and the invariant expected by the event type's contract (lines 15–16) and that maintained by the body (lines 24–25) are the same.

3.4 Direct Interference: Combination

Combination handlers can evaluate the **invoke** expression any number of times. In AspectJ, this would be equivalent to one or more calls to **proceed** in an around advice, guarded by some condition or in a loop. A combination handler is typically useful for implementing functionalities like fault tolerance. We show an example of a combination handler in Figure 8. In this example, we extend figures in the drawing editor to have colors. This is done by adding a field color to class `Fig` and by providing a method `setColor` for picking the color of the figure. The class `Color` is not shown in the listing. It provides a method `nextCol` to get the next available color.

To illustrate combination, let us consider the requirement that "each figure should have a unique color". To implement this requirement, an event type `ClChange` is declared as an abstraction of events representing colors changes. The method `setColor` changes colors so it announces the event `ClChange` on lines 5–7. The body of this announce expression contains the code to obtain the next color (line 6). The handler `Unique` implements the uniqueness requirement by storing already used colors in a hash

```
1  class Fig {
2   Color c;
3   int colorFixed;
4   Color setColor(){
5    announce ClChange(this){
6     this.c = c.nextCol()
7    }
8   }
9  }
10 Color event ClChange{
11  Fig fe;
12  provides fe!=null
13  requires {
14   while(fe.colorFixed==0){
15    invoke(next);
16    either{
17     preserve fe != null
18    } or {
19     preserve fe != null
20    }
21   }
22  }
23 }
```

```
24 class Unique {
25  HashMap colors;
26  when ClChange do check;
27  Color check(thunk Color rest,
28                    Fig fe){
29   while(fe.colorFixed==0){
30    invoke(rest);
31    if(colors.get(fe.c)!=null){
32     refining preserve fe!=null{
33      colors.put(fe.c);
34      fe.colorFixed = 1;fe.c
35     }
36    }else{
37     refining preserve fe!=null{
38      fe.c
39     }
40    }
41   }
42  }
43 }
```

Figure 8: Specifying combination with a translucid contract

table (`colors`). The field `colorFixed` is also added to figure class to show that a unique color has been chosen and fixed for the figure. The initial value of this field is zero. When the handler method `check` is run it checks `colorFixed` to see if a color has been chosen for figure or not, if not it then invokes the event body generating the next candidate color for the figure. If this color is already used, checked by looking it up in the hash table, event body is invoked again to generate the next candidate color. Otherwise, the current color is inserted in the hash table and `colorFixed` is set to one.

The specification for the event type `ClChange` documents that a combination handler will be run when this event is announced. This specification makes use of our novel feature, the choice feature, on line 16–20. To correctly refine this specification, a handler can either have a corresponding **`if`** expression at the corresponding place in its body or it may have an expression that runs unconditionally and refines the **`either`** block or the **`or`** block in the specification. By analyzing the specification, specially by while loop revealed in the specification, the base code developers can understand that the handlers that run when the event `ClChange` is announced may run the original event body multiple times. They are, however, not aware of the concrete details of such handlers, thus those details remain hidden. Since the handler `Unique`'s body structurally matches the specification, it correctly refines the specification.

3.5 More Expressive Control Flow Properties

Rinard *et al.*'s control flow properties are only concerned about calls to **`invoke`**. Their proposed analysis technique can decide which class of interference and category of control effects each isolated advice belongs to [44]. However, it could not be used to analyze possibility of two or more control flow paths each of which is an augmentation, however, each path maintains a different invariant. Figure 9 illustrates such an example. This example is adapted from the work of Khatchadourian and Soundarajan [26].

The class `Fig` not shown here is the same as in Figure 1. Khatchadourian and Soundarajan [26] implement an additional requirement that "a point should always be visibly distinguished from the origin". To implement this requirement a scaling factor `s` is added as a field to the class `Point` (line 2). This factor is initially set to 1 (line 5). The additional requirement is implemented as the class `Scaling`. The handler method `scale` in this class is run when event `Moved` is announced. The handler method ensures that if the point is close enough to the origin (vicinity condition) to visibly distinguish it from the origin the scaling factor is set to 10. Thus the scaling factor only gets two values 1 or 10. The vicinity condition is true if point's x and y coordinates are less than 5.

```
1  class Point extends Fig{
2   int x; int y; int s;
3   Point init(int x,int y){
4    this.x=x; this.y=y;
5    this.s=1; this
6   }
7   int getX(){this.x*this.s}
8   int getY(){this.y*this.s}
9   Fig move(int x, int y){
10   announce Moved(this){
11    this.x=x;this.y=y; this
12   }
13  }
14 }
15 Fig event Moved{
16  Point p;
17  provides p!=null
18  requires{
19   invoke(next);
20   if(p.x<5&& p.y<5){
21    establish p.s==10
22   } else {
23    establish p.s==1
24   }
25  }
26 }
27 class Scaling extends Object{
28  when Moved do scale;
29  Fig scale(thunk Fig rest, Point p){
30   invoke(rest);
31   if(p.x<5 && p.y<5){
32    refining establish p.s==10{
33     p.s=10; p
34    }
35   } else {
36    refining establish p.s==1{
37     p.s == 1; p
38    }
39   }
40  }
41 }
```

Figure 9: Expressive Control Flow Properties Beyond [44]

The assertions we want to validate in this example are as follows: (i) all of the handlers are after augmentation handlers, (ii) the value of scaling factor *s* is either 1 or 10, and (iii) the scaling factor s it is set to 10 if and only if the vicinity condition holds. Rinard *et al.*'s proposal could be used to verify (i) and a behavioral contract can specify (ii) but none of them could specify (iii), whereas our approach can. On lines 19–24 is a specification that conveys to the developers of the class `Point` that a conforming handler method will satisfy all three assertions above.

In summary, in this section we have shown that translucid contracts allow us to specify control flow interference between subject and observers. Specified interference patterns are enforced automatically through refinement. We are able to specify and enforce control interference properties proposed by Rinard *et al.*. There are more sophisticated control flow interplay patterns which could not be enforced by the previous work on design by contract for aspects whereas it could be specified as translucid contracts.

4. APPLICABILITY TO OTHER APPROACHES FOR AO INTERFACES

In this section, we discuss the applicability of our technique to other approaches for AO interfaces. As discussed previously, there are several competing and often complementary proposals

for AO interfaces. For example, Kiczales and Mezini's aspect-aware interfaces (AAI) [29], Gudmundson and Kizales pointcut interfaces [20], Sullivan *et al.*'s crosscutting interfaces (XPIs) [49], Aldrich's Open Modules [1], Steimann *et al.*'s joinpoint types [47], and Rajan and Leavens's event types [41]. We have tried out several of these ideas and our approach works beautifully. Since Steimann *et al.*'s joinpoint types [47] and Hoffman and Eugster's explicit joinpoints (EJP) are similar in spirit to Rajan and Leavens's event types [41], which we have already discussed in previous sections, we do not present our adaptation to these ideas here. Rather we focus on the AspectJ implementation of the XPI approach [52] and Aldrich's Open Modules [1] that are substantially distinct from event types [41, Fig. 10].

4.1 Translucid Contracts for XPIs and AAIs

Sullivan *et al.* [49] proposed a methodology, that they call crosscut programming interface (XPI) for aspect-oriented design based on design rules. The key idea is to establish a design rule interface which serves to decouple the base design and the aspect design. These design rules govern exposure of execution phenomena as joinpoints, how they are exposed through the joinpoint model of the given language, and constraints on behavior across joinpoints (e.g. provides and requires conditions [52]). XPIs prescribe rules for joinpoint exposure, but do not provide a compliance mechanism. Griswold *et al.* have shown that at least some design rules can be enforced automatically using AspectJ's features [52]. Current proposals for XPIs, however, all use behavioral contracts [49]. As shown previously, use of behavioral contracts, limits the expressiveness of the assertions which could be made using XPI. Behavioral contracts could not reveal implementation details which might be needed for some assertions [12].

```
1  class Fig extends Object{ int fixed; }
2  aspect XChanged {
3   pointcut joinpoint(Fig fe): target(fe)
4              && call(void Fig+.set*(..));
5   provides fe != null
6   requires {
7    if(fe.fixed == 0){
8     proceed();
9     preserve fe==old(fe)
10   } else {
11    preserve fe==old(fe)
12   }
13  }
14 }
15 aspect Enforce {
16  Fig around (Fig fe):
17      XChanged.joinpoint(fe){
18   if(fe.fixed == 0){
19    proceed()
20   } else {
21    refining preserve fe==old(fe){
22      fe
23    }
24   }
25  }
26 }
```

Figure 10: Applying translucid contract to an XPI

In this section, we show that translucid contracts can also be applied to enable expressive assertions about aspect-oriented programs that use the XPI approach. We also discuss changes in the refinement rules that is needed to verify such programs. To illustrate consider the example from Section 3.2, where constraint on movement of figures is implemented as an XPI and an aspect. An XPI typically also contains a description of scope, which we omit here. In the context of XPIs, the language for expressing translucid contract is slightly adapted to use **`proceed`** instead of **`invoke`** on line 8. Other than this change, our syntax for translucid contracts works right out-of-the-box.

Unlike translucid contracts for event types in Ptolemy, where the contract is thought to be attached to the type, in the AspectJ's version of XPI contracts are thought to be attached to the pointcut declaration, e.g. the contract on lines 5–13 is attached to the pointcut on lines 3–4. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract can only use `fe`.

Our proposal for verifying refinement also needs only minor changes. Unlike Ptolemy, where the event types of interest are specified in the binding declarations, in AspectJ's version of XPI, aspects reuse the pointcut declarations from the XPI in the advice declaration (lines 16–17). Our refinement rules could be added here in the AO type system. So for an advice declaration to be well-formed, its pointcut declaration must be well-formed, the advice body must be well-formed, and the advice body must refine the translucid contract of the pointcut declaration. This strategy works for basic pointcuts, for compound pointcuts constructed using rules such as (`pcd and pcd'` or `pcd` **`or`** `pcd'`), where both `pcd` and `pcd'` are reused from different XPIs and thus may have independent contracts more complex refinement rules will be needed, which we have not explored in this paper.

Joinpoint interfaces like XPI could be computed from the implementation rather than being explicitly specified given whole-program information. Kiczales and Mezini [29] follow this approach to extract aspect-aware interfaces (AAI). A detailed discussion of the trade-offs of such interfaces is the subject of previous work [42, 49]. However, an important property of AAIs is that advised joinpoints contain the details of the advice. An example based on the example in Figure 10 is shown in Figure 11. The extracted AAI for the method `setX` is shown on lines 3-4. An adaptation of this extraction to include translucid contracts will be to carry over the contract from the pointcut to the joinpoint shadow as shown on lines 5–13.

```
1  Point extends Fig {
2   int x; int y;
3   Fig setX(int x): Update -
4    after returning Update.joinpoint(Fig fe)
5     provides fe != null
6     requires {
7      if(fe.fixed == 0){
8        proceed();
9        preserve fe==old(fe)
10     } else {
11       preserve fe==old(fe)
12     }
13    }
14  /* body of setX */
15  }
```

Figure 11: Applying translucid contract to an AAI

The syntax and refinement rules similar to XPI are applicable here. Like AAI annotations that provide developers of `Point` with information about potentially advising aspects, added contract would provide developers of `Point` with richer abstraction over the aspect's behavior. Similar ideas can also be applied to aspect-oriented development environments such as AJDT, which provide AAI-like information at joinpoint shadows in an AO program.

4.2 Translucid Contracts for Open Modules

Aldrich's proposal on Open Modules [1] is closely related to Ptolemy's quantified, typed events [41]. Open modules allows a class developer to explicitly expose pointcuts for behavioral modifications by aspects, which is similar to signaling events using the **announce** expressions in Ptolemy. The implementations of these pointcuts remain hidden from the aspects. As a result, the impact of base code changes on the aspect is reduced. However, quantification in Ptolemy is more expressive compared to Open Modules. In open modules, each explicitly declared pointcut has to be enumerated by the aspect for advising. On the other hand, Ptolemy's quantified, typed events significantly simplify quantification. Instead of manually enumerating the joinpoints of interest, one can use the name of the event type for implicit non-syntactic selection of joinpoints. This affects applicability of translucid contracts to Open Modules.

```
1  module FigModule {
2   class Fig;
3   friend Enforce;
4   expose: target(fe) &&
5    call(void Fig+.set*(..));
6    provides fe != null
7    requires{
8     if(fe.fixed == 0){
9      proceed();
10     preserves fe == old(fe)
11    } else {
12     preserves fe == old(fe)
13    }
14   }
15 }
16 aspect Enforce {
17  Fig around (Fig fe): target(fe) &&
18   call(void Fig+.set*(...)){
19    if(fe.fixed == 0){
20     proceed()
21    } else {
22     refining preserve fe==old(fe){
23     fe
24    }
25   }
26  }
27 }
```

Figure 12: Applying translucid contract to Open Modules [39]

To show the applicability of translucid contracts to Open Modules, we revisit the narrowing example from Section 3.2. Figure 12 shows the implementation of the same scenario using Open Modules. In implementing the example, we use the syntax from the work of Ongkingco *et al.* [39] to retain similarity with other examples in this work. In the listing constraints on the movement of figure is encapsulated in the module `Enforce`. Open module `FigModule` exposes a pointcut of **`class`** `Fig` on lines 4–5, marked by the keyword **`expose`**. Exposed pointcut is advisable only by the **`friend`** aspect `Enforce`. Transcluid contract on lines 6–14 states the behavior of interaction between specified friend aspect and the exposed pointcut. The adaptations in the syntax of contracts are the same as in the case of XPI discussed in Section 4.1.

Like contracts in XPI, contracts in Open Modules are attached to a pointcut declaration, e.g. the contract on lines 6–14 is attached to the exposed pointcut defined on lines 4–5. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract on lines 6–14 can only use the variable `fe`.

Proposed verifying refinement rules need to be modified slightly as well. In Ptolemy, event type of interest is specified in the binding declaration whereas in AspectJ's version of Open Modules, aspects could not reuse pointcuts exposed by an Open Module and need to enumerate the pointcut in the advice declaration again (lines 17–18). Our refinement rules could be added here in AO type system. Well-formedness of basic and compound pointcuts follow the same rules laid out in Section 4.1.

This example illustrates how our approach might be used as a specification and verification technique for Open Modules. The only challenge that we saw in this process was to match an aspect's pointcut definition with the open module's pointcut definition to import its contract for checking refinement. Like translucid contracts for Ptolemy, in the case of Open Modules specification serves as a more expressive documentation of the interface between aspects and classes.

5. RELATED IDEAS

There is a rich and extensive body of ideas that are related to ours. Here, we discuss those that are closely related under three categories: contracts for aspects, proposals for modular reasoning, and verification approaches based on grey box specifications.

Contracts for Aspects. This work is closest in spirit to the work on crosscut programming interfaces (XPIs) [23, 52]. XPIs also allow contracts to be written as part of the interfaces as **`provides`** and **`requires`** clauses. Similar to translucid contracts, the **`provides`** clause establishes a contract on the code that announces events, whereas the **`requires`** clauses specifies obligations of the code that handles events. However, the contracts specified by these works are mostly informal and cannot be automatically checked. Furthermore, these works do not describe a verification technique and contracts could be bypassed.

Skotiniotis and Lorenz [33, 46] propose contracts for both objects and aspects in their tool *Cona*. Rinard *et al.* [44] classify the interaction of advice and method into direct and indirect interactions. Direct interactions focus on control flow elements while indirect interactions are concerned about data elements. Each of direct and indirect interactions are further categorized into different classes of interactions. They have developed an analysis system that categorizes aspects and method interactions. Their classification and analysis system serves reasoning purposes. As an analysis system, it expect developers to enforce desired properties by informing them about classes that each aspect-method interaction belongs to. There is no specification/enforcement mechanism supported in their approach. Zhao and Rinard [55] propose Pipa as a behavioral specification language for AspectJ. Pipa supports specification inheritance and specification crosscutting. It relies on textual copying of specifications for specification inheritance and syntactical weaving of specification for specification crosscutting. Annotated AspectJ program with JML-like Pipa's specifications is transformed to JML and Java code. JML-based verification tool could be used later to enforce specified behavioral constraints. All of these ideas use behavioral contracts and thus may not be used to reason about control effects of advice.

Modular Reasoning. There is a large body of work on modular reasoning about AO programs on new language designs [1, 14, 17, 21, 25], design methods [23, 30, 32, 52], and verification techniques [3, 24, 31, 40]. Our work complements ideas in the first and the second category and can use ideas in the third category for improved expressiveness. Compared to work on reasoning about implicit invocation [4, 8, 19, 54], our approach based on structural refinement is significantly lightweight. Furthermore, it accounts for quantification that these ideas do not.

Oliveira *et al.* [38] introduce a non-oblivious core language with explicit advice points and explicit advice composition requiring effects modeled as monads to be part of the component interfaces. Their statically typed model could enforce control and data flow in-

terference properties. Their work shares commonalities with ours in terms of explicit interfaces having more expressive contracts to state and enforce the behavior of interactions. However, it is difficult to adapt their ideas built upon their non-AO core language, to II, AO, and Ptolemy as they do not support quantification.

Hoffman and Eugster Explicit Joint Points [21] and Steimann *et al.*'s Joint Point Types [47] share similar spirit with Rajan and Leavan's event types [41]. Although Steimann *et al.* proposed informal behavioral specification, but there is no explicit notion of formally expressed and enforced contracts, stating interactions behavior, in any of these approaches.

Grey Box Specification and Verification. This work builds upon previous research on grey box specification and verification [12]. Among others, Barnett and Schulte [6, 7] have considered using grey box specifications written in AsmL [5] for verifying contracts for .NET framework, Wasserman and Blum [53] also use a restricted form of grey box specifications for verification, Tyler and Soundarajan [51] and most recently Shaner, Leavens, and Naumann [45] have used grey box specifications for verification of methods that make mandatory calls to other dynamically-dispatched methods. Rajan *et al.* have used grey box specifications to enable expressive assertions about web-services [43]. Compared to these ideas, our work is the first to consider grey box specifications as a mechanism to enable modular reasoning about code that announces events from the code that handles events, which is a common idiom in AO and II languages.

6. FUTURE WORK

Having laid out the basic ideas behind translucid contracts we now turn to the remaining work necessary to incorporate translucid contracts in Ptolemy and other AO languages. First task would be to define a precise formalization of when translucid contracts are refined by the handler body and second would be give definition of how reasoning about **`announce`** expressions proceeds. We discuss our ideas to solve these problems in some detail below.

6.1 Checking Contract Refinement

In Ptolemy a module announcing an event of type p may assume that all handler methods that run when p is announced refine the translucid contract of p. Each such handler method then in turn must guarantee that it correctly refines the contract. Informally, showing refinement between a contract and a handler method builds on the notion of structural refinement from the work of Shaner *et al.* [45]. New ideas will be in the refinement of **`invoke`** and **`either`** `{..}` **`or`** `{..}` expressions. The contract may include specification notations (se) whereas implementations can only contain executable code, built solely from the program constructs (e) described in Figure 3. A handler method's implementation refines a translucid contract if it meets two criteria: first, that the code shares its structure with that given in the specification and second, that the body of every **`refining`** expression obeys the specification it is refining.

6.2 Reasoning about Announce Expressions

A technique for verification of the code that announces events is an important future work. The basic plan is to use the copy rule [35] substituting the abstract program that is the specification of the event type for the announce expression. This will be sound, due to structural refinement.

The technical difficulty is that handlers are essentially mutually recursive, since they can use **`invoke`** expressions to proceed to the next handler that applies to the event. Thus in the verification, it will be difficult to know, statically, how many times to substitute (unfold) the abstract program for **`invoke`** when reasoning about an announce expression. One possibility is to use predicates that describe the state of the active object list, so that in the verification one can do a case split, based on what handlers actually do apply to a particular announce expression at that point in the program.

7. CONCLUSION

Many recent proposals for aspect-oriented interfaces [1, 20, 21, 29, 41, 47, 49] show promise towards improving modularity of AO programs. An important benefit of these proposals is that they provide a syntactic location to specify contracts between the advised and the advising code. However, behavioral contracts [49, 55] are largely insufficient to reason about potentially important control-flow related properties of the advising code.

We show that translucid contracts that are based on grey box specifications [10, 11] are useful for understanding and enforcing such properties for Ptolemy programs. A handler method conforms with such contract, if its body refines the contract. All handler methods for an event type p are required to conform with the contract for p. We show how verification proceeds for code announcing events. Checking the handlers and the announce expressions only require module-level analysis. We also demonstrate the applicability of translucid contracts to other type of AO interfaces.

Besides direction already discussed in Section 6, adding translucid contracts to other AO compilers, integrating it with rich specification features in JML, and trying out larger examples would also be part of the future activities.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP'05*.

[2] R.-J. Back and J. von Wright. *Refinement Calculus: A Systematic Introduction*. Graduate Texts in Computer Science. Springer-Verlag, 1998.

[3] P. F. Baldi, C. V. Lopes, E. J. Linstead, and S. K. Bajracharya. A theory of aspects as latent topics. In *OOPSLA*, pages 543–562, 2008.

[4] L. Baresi, C. Ghezzi, and L. Mottola. On accurate automatic verification of publish-subscribe architectures. In *ICSE'07*.

[5] M. Barnett and W. Schulte. The ABCs of specification: AsmL, Behavior, and Components. *Informatica*, 25(4):517–526, November 2001.

[6] M. Barnett and W. Schulte. Spying on components: A runtime verification technique. In *SAVCBS*, 2001.

[7] M. Barnett and W. Schulte. Runtime verification of .NET contracts. *JSS*, 65(3):199–208, March 2003.

[8] J. S. Bradbury and J. Dingel. Evaluating and improving the automatic analysis of implicit invocation systems. In *ESEC/FSE'03*.

[9] M. Büchi. Safe language mechanisms for modularization and concurrency. Technical Report TUCS Dissertations No. 28, Turku Center for Computer Science, May 2000.

[10] M. Büchi and E. Sekerinski. Formal methods for component software: The refinement calculus perspective. In *WCOP'97*.

[11] M. Büchi and W. Weck. A plea for grey-box components. Technical Report 122, Turku Center for Computer Sc., 1997.

[12] M. Büchi and W. Weck. The greybox approach: When blackbox specifications hide too much. Technical Report 297, Turku Center for Computer Science, August 1999.

[13] C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.

[14] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. In *SPLAT'03*.

[15] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *SCP*, 63(3):321–374, 2006.

[16] C. Clifton, G. T. Leavens, and J. Noble. Ownership and effects for more effective reasoning about Aspects. In *ECOOP '07*.

[17] D. S. Dantas and D. Walker. Harmless advice. In *POPL'06*.

[18] M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, 1999.

[19] D. Garlan, S. Jha, D. Notkin, and J. Dingel. Reasoning about implicit invocation. In *FSE'98*.

[20] S. Gudmundson and G. Kiczales. Addressing practical software development issues in AspectJ with a pointcut interface. In *Advanced Separation of Concerns*, 2001.

[21] K. J. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ'07*.

[22] A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*.

[23] K. J. Sullivan *et al.* Information hiding interfaces for aspect-oriented design. In *ESEC/FSE'05*.

[24] S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL'4*.

[25] A. Kellens, K. Mens, J. Brichau, and K. Gybels. Managing the evolution of aspect-oriented software with model-based pointcuts. In *ECOOP '06*.

[26] R. Khatchadourian and N. Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *SPLAT'07*.

[27] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP'01*.

[28] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP'07*.

[29] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE'05*.

[30] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP'05*.

[31] S. Krishnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *FSE'04*.

[32] C. V. Lopes and S. K. Bajracharya. An analysis of modularity in aspect oriented design. In *AOSD*, pages 15–26, 2005.

[33] D. H. Lorenz and T. Skotiniotis. Extending design by contract for aspect-oriented programming. *CoRR*, abs/cs/0501070, 2005.

[34] D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *TSE*, 21(4), Apr 1995.

[35] C. Morgan. Procedures, parameters, and abstraction: separate concerns. *Sci. Comput. Program.*, 11(1):17–27, 1988.

[36] C. Morgan. *Programming from Specifications: Second Edition*. Prentice Hall International, Hempstead, UK, 1994.

[37] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *SCP'87*.

[38] B. Oliveira, T. Schrijvers, and W. R. Cook. Effectiveadvice: Disciplined advice with explicit effects. In *AOSD'10*.

[39] N. Ongkingco, P. Avgustinov, J. Tibble, L. Hendren, O. de Moor, and G. Sittampalam. Adding Open Modules to AspectJ. In *AOSD'6*.

[40] K. Ostermann. Reasoning about aspects with common sense. In *AOSD*, 2008.

[41] H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP'08*.

[42] H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *TOSEM*, 2008.

[43] H. Rajan, J. Tao, S. M. Shaner, and G. T. Leavens. Tisa: A language design and modular verification technique for temporal policies in web services. In *ESOP'09*.

[44] M. Rinard, R. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *FSE'04*.

[45] S. M. Shaner, G. T. Leavens, and D. A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *OOPSLA'07*.

[46] T. Skotiniotis and D. H. Lorenz. Cona: Aspects for contracts and contracts for aspects. In *OOPSLA'04*.

[47] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM*, 20(1), 2010.

[48] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'05*.

[49] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM*, 20(2), 2009.

[50] T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In *SPLAT'03*.

[51] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES'03*.

[52] W. G. Griswold *et al.* Modular software design with crosscutting interfaces. *IEEE Software'06*.

[53] H. Wasserman and M. Blum. Software reliability via run-time result-checking. *J. ACM*, 44(6):826–849, 1997.

[54] H. Zhang, J. S. Bradbury, J. R. Cordy, and J. Dingel. Using source transformation to test and model check implicit-invocation systems. *SCP'06*.

[55] J. Zhao and M. Rinard. Pipa: A behavioral interface specification language for AspectJ. In *FASE'03*.