Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces

Mehdi Bagherzadeh  
*Iowa State University*, mbagherz@iastate.edu

Hridesh Rajan  
*Iowa State University*

Gary T. Leavens  
*Iowa State University*

Sean L. Mooney  
*Iowa State University*, smooney@iastate.edu

Follow this and additional works at: [http://lib.dr.iastate.edu/cs_techreports](http://lib.dr.iastate.edu/cs_techreports)  
Part of the [Other Computer Sciences Commons](http://lib.dr.iastate.edu/cs_techreports)

**Recommended Citation**  
[http://lib.dr.iastate.edu/cs_techreports/260](http://lib.dr.iastate.edu/cs_techreports/260)

This Article is brought to you for free and open access by the Computer Science at Digital Repository @ Iowa State University. It has been accepted for inclusion in Computer Science Technical Reports by an authorized administrator of Digital Repository @ Iowa State University. For more information, please contact digirep@iastate.edu.
Translucid Contracts: Expressive Specification and Modular Verification for Aspect-oriented Interfaces

Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens and Sean Mooney

TR #10-02a
Initial Submission: March 1, 2010.
Revised: July 1, 2010.

Keywords:
Aspect-oriented programming, Modular reasoning, Modular verification, Interfaces, Design by contract

CR Categories:
D.2.10 [Software Engineering] Design
D.1.5 [Programming Techniques] Object-Oriented Programming
D.2.2 [Design Tools and Techniques] Modules and interfaces, Object-oriented design methods
D.2.3 [Coding Tools and Techniques] Object-Oriented Programming
D.2.4 [Software/Program Verification] Programming by contract, Validation
D.3.3 [Programming Languages] Language Constructs and Features - Control structures

Copyright (c) 2010, Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens and Sean Mooney.
Submitted for publication.

Laboratory for Software Design
Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA
Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces

Mehdi Bagherzadeh*, Hridesh Rajan*, Gary T. Leavensθ and Sean Mooneyβ

*Iowa State University, Ames, IA, USA
θUniversity of Central Florida, Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

As aspect-oriented programming techniques become more widely used, their use in critical systems, including safety-critical systems such as aircraft and mission-critical systems such as telephone networks, will become more widespread. However, careful reasoning about aspect-oriented code seems difficult with standard specification techniques. The difficulty stems from the difficulty of understanding control effects, such as advice that does not proceed to the original join point, because most common specification techniques do not make it convenient to specify such control effects. In this work we give a simple and understandable specification technique, which we call translucid contracts, that not only allows programmers to write modular specifications for advice and advised code, but also allows them to reason about the code’s control effects. We show that translucid contracts support modular verification of typical interaction patterns used in aspect-oriented code. We also show that translucid contracts allow interesting control effects to be understood and enforced. Our proposed specification and verification approach is proved sound.

1. INTRODUCTION

Reasoning about aspect-oriented (AO) programs that use pointcuts and dynamic advice, as found in AspectJ programs, often seems difficult, due to two fundamental problems:

1. Join point shadows, i.e., places in the code where advice may apply, occur very frequently1, and at each join point shadow reasoning about the effect of the code at that place in the code must take into account the effects of all applicable advice.

2. The control effects of advice must be understood in order to reason about a program’s control flow and how advice might interfere with the execution of other advice.

As an example of the first problem, consider code such as that in Figure 1. In that figure, assuming that x and y are fields, there are at least 8 join point shadows, including the 5 method calls, the writes of x and y, and the read of x.

Figure 1: AspectJ code illustrates density of join point shadows.

1 x = o1.m1(a.e1(), b.e2());
2 y = o2.m2(c.e3(), x);

1 aspect Overriding {
2  void around () : call (void Account+.* (..)) {...}
3 }
4 aspect Authentication {
5  void around () : call (void Account+.* (..)) {...}
6 }

1 x = o1.m1(a.e1(), b.e2());
2 y = o2.m2(c.e3(), x);

Figure 2: AspectJ code illustrates advice interference.

As an example of the second problem consider two aspects in Figure 2 that both advise the same set of join points and contain an around advice. To understand the control flow at the join points matched by these aspects a developer must understand the control flow of both pieces of advice. Furthermore, to understand the behavior at such points one must also understand the control flow of all other advice that may advise the same join points.

1.1 Previous Work on These Problems

One way of solving the first problem is to limit where advice may apply, for example, by using some form of explicit base-advice interface (AO interface), e.g. crosscutting interfaces (XPIs), open modules, etc [2–6]. However, none of these works has investigated the effectiveness of such interfaces towards enabling a design by contract (DBC) methodology for aspect-oriented software development (AOSD). This is not to be confused with DBC using AOP, where the advice construct is used to enforce contract of a method. Rather we speak of the contract between aspects and the base code. Design by contract (DBC) methodologies for AOSD have been explored before [2,7,8], however, existing work relies on black box 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. To illustrate, consider

Submitted for publication. Copyright retained by the authors.
the snippets shown in Figure 3 from the canonical drawing editor example with functionality to draw points, lines, and a display updating functionality.

Figure 3 uses crosscutting interface (XPI) [2] as the AO interface. XPI is a design rule interface which governs the exposure of join points and constrains the behavior across the exposed join points. In Figure 3 XPI Changed declares a black box contract on lines 12–13 for its pointcut description (PCD) named jp, lines 10–11. This PCD is used in the advice on lines 16–20. While the declaration of the pointcut is a sensible place to document the behavior of advice in AspectJ, there would be nothing at a join point shadow, such as a call to setX, that would indicate to a programmer reasoning about such a call that advice is being applied. This relates to the first of our reasoning problems.

To illustrate the second of our reasoning problems, note that the black box behavioral specification on lines 12–13 does not specify the control effects of the advice. For example, with just the behavioral specification of the XPI Changed given, one cannot determine whether a call such as p.setX(3) will proceed to execute the body of setX, and thus whether such a call will always set the current x coordinate of p to its argument (3). Such assertions are important for reasoning, which depends on understanding the effect of composing the aspect modules with the base code [2, 10]. That is, the contract does not specify whether the advice must always proceed. Ideas from Zhao and Rinard’s Pipa language [8], if applied to AO interfaces help to some extent. However, as we discuss in greater detail in Section 6, Pipa’s expressiveness beyond simple control flow properties is limited.

Another problem with such black box behavioral contracts is that they do not help with effectively reasoning about the effects of aspects on other. Consider another example concern, say Logging, which writes a log file at the same join points declared in the PCD in aspect Changed. For this concern different orders of composition with the Update concern in Figure 3 could lead to different results. (In AspectJ declare precedence can be used to enforce an ordering on aspects and the application of their advice.) Suppose line 18 of Figure 3 was omitted; that is, suppose that Update’s advice did not proceed. In that case, if Update were to run first, followed by Logging, then the evaluation of Logging would be skipped. Conversely, Logging would work (i.e., it would write the log file) if the aspects were composed in the opposite order. An aspect developer can not, by just looking at the black box behavioral contract of the AO interface, reason about the composition of such aspects. Rather a developer must be aware of the control effects of the code in all composed aspects. Furthermore, if any of these aspect modules changes (i.e., if their control effects change), one must reason about every other aspect that applies at the same join points.

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 [2]. But 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 Solution: Translucid Contracts

The main contribution of this work is the notion of translucid contracts for AO interfaces, which is based on grey box specification [11]. 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 expressions. 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 [12], but we use a restricted form of refinement that requires structural similarity, to allow specification of control effects.

To illustrate, consider the translucid contract shown in Figure 4. Figure 4 uses a proposal for aspect interfaces, event types, promoted by our previous work on the Ptolemy language [6]. In Ptolemy, events are explicitly announced which mitigates our first problem, as reasoning about events only needs to happen at program points where events are explicitly announced (such as lines 5–7). Ptolemy programs declare event types, which are abstractions over concrete events in the program. Lines 11–19 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 12, the changing figure, named fe, is made available. Concrete events of this type are explicitly and declaratively created using announce expressions as shown on lines 5–7. Like Eos [13, 14], Ptolemy doesn’t distinguish between aspects and classes. On lines 20–29 is the Ptolemy-equivalent of the Update aspect in Figure 3. The Update class has a binding declaration on line 28 that says to run the method update when events of type Changed are signaled. Ptolemy also provides dynamic registration using register, as shown on line 21, which activates the current instance of the Update class as an observer for events.

We have added an example translucid contract to the interface, event type Changed, on lines 13–18. Contrary to the black box behavioral contract, internal states of the handler methods (which correspond to advice) that run when the event Changed is announced (this corresponds to a join point occurrence) are exposed in the translucid contract. In particular, any occurrence of invoke expression (which is like AspectJ’s proceed) 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 meth-

---

2This limitation of black box behavioral specifications was discussed in a preliminary version of this paper that was presented at the FOAL 2010 workshop [9].
Figure 4: A translucid contract for aspect interfaces using Ptolemy [6] as the implementation language. See Section 2.1 for syntax.

- 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 to state these contracts. We use our previous work on the Ptolemy language [6] for this discussion. However, as we show in Section 5, our basic ideas are applicable to other aspect-oriented programming models. We first present Ptolemy’s programming features and then describe its specification features.

2.1 Program Syntax

Ptolemy is an object-oriented (OO) language with support for declaring, announcing, and registering with events much like implicit-invocation (II) languages. 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 “quantification” in aspect-oriented languages such as AspectJ. The formally defined OO subset of Ptolemy 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 5 and explained below. A Ptolemy program consists of zero or more declarations, and a “main” expression (see Figure 4). Declarations are either class declarations or event type declarations.

Figure 5: Ptolemy’s syntax [6], with refining expressions and contracts added

3
2.1.1 Declarations

We do not allow nesting of decls. Each class has a name (c) and names its superclass (d), and may declare fields (field) and methods (method). Field declarations are written with a class name, giving the field's type, followed by a field name. Method headers 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 (m) [6]. An example is shown in Figure 4, which contains a binding on line 28. This binding declaration tells Ptolemy to run the method update whenever events of type Changed are announced. II terminology calls such methods handler methods.

An event type (event) declaration has a return type (t), 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 [6]. An example is given in Figure 4 on lines 11–19. In writing examples of event types, as in Figure 4, 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.

2.1.2 Expressions

The formal definition of Ptolemy is given as an expression language [6]. 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 (e1.f = e2), a definition block (t var = e1; e2), and sequencing (e1; e2). Their semantics and typing is fairly standard [4, 6] and we encourage the reader to consult [6].

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. Only active objects in this list are capable of advising events. For example line 21 of Figure 4 is a method that, when called, will register the method's receiver (this). The expression announce p(e)(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 [6]. 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 next 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. The type thunk t ensures that the value of the corresponding actual parameter is an event closure with return type t, and hence t is 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 4 line 22). Event closures are never stored; they are only constructed by the semantics and passed to the handler methods.

There is one additional 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 6. In this figure, all non-terminals that are used but not defined are the same as in Figure 5.

\[
\text{contract ::= requires sp \ assumes \ (se) \ ensures \ sp} \\
sp ::= \text{sp \ requires \ sp \ assumes \ sp \ spec} \\
sp ::= \text{null \ | \ new \ c(se) \ | \ rest \ sp \ | \ if \ sp \ | \ while \ sp \ | \ cast \ c(se) \ | \ invoke \ (se) \ | \ declare \ (se) \ | \ announce \ (se) \ | \ refine \ spec} \\
\text{requires \ sp \ ensures \ sp} \\
\text{requires \ sp \ assumes \ sp \ spec} \\
sp ::= \text{null \ | \ new \ c(se) \ | \ rest \ sp \ | \ if \ sp \ | \ while \ sp \ | \ cast \ c(se) \ | \ invoke \ (se) \ | \ announce \ (se) \ | \ refine \ spec}
\]

Figure 6: Syntax for writing translucid contracts

A contract is of the form requires sp, assumes {se} ensures sp2. Here, sp1 and sp2 are specification predicates as defined in Figure 6 and the body of the contract se is an expression that allows some extra specification-only constructs (such as the choice construct either se1 or se2). In an event specification, the predicate sp1 is the precondition for event announcement, and sp2 is the postcondition of the event announcement. The specification expression se is the abstract algorithm describing conforming handler methods. The invoke expressions must be revealed in se and the variables that could be named in se are only context variables. 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 4 the method update on lines 22–27 must refine the contract of the event type Changed on lines 13–18.

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 [15, 18]. The most general form of specification expression is requires sp1 ensures sp2, where sp1 is a precondition expression and sp2 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 sp1, will terminate in a state that satisfies sp2 [15, 18]. In examples we use the following syntactic sugars: preserves sp for requires sp ensures sp, and establishes sp for requires 1 ensures sp [18]. 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 {se1} or {se2}) are place holders in the specification that express the event close passed to a handler, the call of an event handler using invoke, and a conditional expression in a conforming handler method, respectively. The choice expression hides and thus abstracts from the concrete condition check in the handler method. For a choice expression either {se1} or {se2} a conforming handler method may contain an expression e1 that refines se1, or an expression e2 that refines se2, or an expression if (e0) {e1} else {e2}, where e0 is a side-effect free expression, e1 refines se1, and e2 refines se2.

3. VERIFICATION OF PROGRAMS WITH TRANSLUCID CONTRACTS

Verifying Ptolemy programs (or AspectJ programs) is different from standard object-oriented (OO) programs in two ways. First, a method in the program under verification may announce events that can cause a set of handlers to run. In AspectJ, this is equivalent to invoking a set of advice at a join point. Second, if the method is a handler it may call invoke that can also cause a set of handlers to run. In AspectJ, this is equivalent to an advice calling proceed that can cause other advice to run.
Therefore, verifying a Ptolemy program with translucid contracts poses two novel technical problems, compared to verifying standard OO programs: (1) verifying that each handler method correctly refines the contract of each event type it handles, and (2) verifying code containing announce and invoke expressions.

A handler method is a method that is statically declared in a binding form in its class to handle events of a given event type. When a binding of the form when p do m appears in a class declaration, then we say that m is a handler method for event type p, e.g., handler method update in Figure 4.

The main novelty of translucid contracts is that both these verification steps can be carried out modularly. By “modularly” we mean that each task can be done using only the code in question, the specifications of static types mentioned in the code, and the specifications of the relevant event types. For a handler, the relevant event type specifications are all the event types that the method is a handler for. For an announce expression, the relevant event type is the one that is being announced. For an invoke expression, which must occur inside a handler method body, it is each event type that the method is a handler for.

3.1 Overview of Key Ideas in Verification

Informally, to verify that each handler method correctly refines the contract of each event type that it handles, we first statically check whether the structure of the handler method matches the structure of the assumes block of the event type. Note that invoke expressions that can override the underlying event body’s execution (join point in AO terms) can only appear inside the handler method. So this check ensures that the control effects of the handler method matches the control effects specified in the translucid contract. At the same time, in our current implementation, we insert runtime assertions that check that the pre- and postconditions required by each event type’s contract are satisfied by the handler method. These two checks ensure that starting with a state that satisfies the event type’s precondition, if a correct handler method is run, it will terminate in a state that satisfies the event type’s postcondition, while ensuring that it produces no more control effects than those mentioned in the event type’s assumes block.

Recall that an announce expression may cause a statically unknown number of handler methods to run, potentially followed by the event body. In AspectJ terms, this is equivalent to running unknown number of pieces of advice, potentially followed by the original join point code. An invoke expression (proceed) works similarly. To verify the code containing an announce expression, we take advantage of the fact that each correct handler method refines the event type’s contract. So the event type’s contract can be taken as a sound specification of the behavior of each handler.

What is interesting and novel about our proposal is that the assumes block for an event type’s translucid contract gives a sound specification of the behavior of an arbitrary number of handlers for that event.

Ignoring concrete details at the moment, imagine that we need a sound specification of the behavior of two handlers for the event type Changed in Figure 4. This can be constructed by taking the assumes block of this event type’s contract and replacing occurrences of all invoke expressions inside it by the same assumes block (we will discuss how to do this shortly). This essentially achieves the effect of inlining the invoke expression (and is similar to unrolling a loop or inlining a recursive call [19]). Notice that construction of this specification only requires access to the event type. Also note that the resulting specification may contain some invoke expressions (as a result of inlining the assumes block). Let us call the constructed specification $S$.

Given the specification $S$ on the behavior of two handlers, we can now (1) reason about the code containing an announce expression as well as (2) the code containing an invoke expression. Again, ignoring concrete details, in the code containing the announce expression we do have access to the event body. So we replace all invoke expressions in $S$ with this event body. As a result, we now have a pure OO specification expression that is a sound specification of this announcement of the event Changed, $S_{\text{ann}}$. This specification expression can be used to reason about the code that contains announce expression. An important property of this step is that we only used the event type’s contract and the code that was announcing events.

To reason about the code that contains invoke expression, once again we start with the specification constructed from event type’s contract, i.e., $S$. Note that the event body must refine the event type pre- and postcondition (to avoid surprising handler methods). So we replace all invoke expression in $S$ expression with the pre- and postcondition of the event type’s contract. This again gives us a pure OO specification expression that is a sound specification of running two handlers and a correct event body, $S_{\text{inv}}$. Similarly, in this step as well, we only used the event type’s contract and the code that contains invoke expression.

In the rest of this section, we describe these verification steps starting with handler refinement.

3.2 Checking Handler Refinement

To enable modular reasoning, all handlers must be correct. A correct handler method in Ptolemy must refine the translucid contract of each event type that the method handles. Checking refinement of such a method is done in a two-step process. First, we statically verify whether the handler method’s body, which is an expression ($e$) is a structural refinement of the translucid contract of the event type, which is a specification expression ($se$). This step is performed as part of type-checking in Ptolemy’s compiler. Second, we verify that handler method satisfies the pre- and postconditions of the event type specification. This is currently checked at runtime (Section 3.4), however, a static approach such as extended static checking [19] could also be applied.

Figure 7 shows the structural refinement process where refinement is checked for each handler method binding. $CT$ is a fixed list of program’s declarations. Rule (CLASS TABLE REF) in Figure 7 checks structural refinement for each handler binding in the program. Rule (CHECK BINDING REF) creates the typing contexts ($\pi, \Pi$) for the specification expression that is the body of the translucid contract and the program expression that is the body of the handler method and uses rules in Figure 8 to check their structural refinement. In structural refinement, specification expressions in the contract are refined by (possibly different) program expressions in an implementation; however, program expression in the contract are refined by textually identical program expressions in the implementation.

A specification expression is refined by a program expression if its subexpressions are refined by corresponding subexpressions of the concrete program expression. Figure 8 shows rules for checking that. There is no rule for register as it is not allowed in an event type specification. Judgement $(\pi, \Pi) \vdash se \subseteq e$ states that specification expression $se$ is refined by program expression $e$ in the specification typing environment $\pi$ and program expression typing environment $\Pi$, which in turn are constructed in the (CHECK BINDING REF) rule.
Fig event Changed{
...
13 requires fe != null
14 assumes{
15 invoke( next );
16 establishes fe==old(fe) 
17 ...
23 invoke( rest );
24 refining establishes fe==old(fe){
25 Display.update(fe); fe
26 }
27 }
28
Refines

Figure 7: Rules for checking structural refinement

| (π, II) ⊢ se ⊑ e, where π, II: specification, program typing contexts, and se: specification expression, e: program expression |
|-----------------|-----------------|-----------------|
| Cases of Spec. Exp. (se) | Refined By (e) | Side Conditions |
| n               | n               | n               |
| var             | var'            | if (var) == II(var') |
| sp; f           | sp; f           | if (π, II) ⊢ sp ⊑ sp |
| sp; null        | sp; !null       | if (π, II) ⊢ sp ⊑ sp |
| sp; l           | sp               | if (π, II) ⊢ sp ⊑ sp |
| sp; l & sp'     | sp; l & sp'     | if (π, II) ⊢ sp ⊑ sp |
| sp = m          | sp = m          | if (π, II) ⊢ sp ⊑ sp |
| sp < n          | sp < n          | if (π, II) ⊢ sp ⊑ sp |
| se₁; se₂        | e₁; e₂          | if (π, II) ⊢ se₁ ⊑ e₂, (π, II) ⊢ se₁ ⊑ e₂ |
| if(se₁)(se₂)    | if(ep)(e₁)      | if (π, II) ⊢ e₁ ⊑ ep, (π, II) ⊢ e₁ ⊑ e₁ |
| else(se₂)       | else(e₁)        | (π, II) ⊢ e₁ ⊑ ep, (π, II) ⊢ e₁ ⊑ e₁ |
| while(se₁)      | while(ep)(e₁)   | if (π, II) ⊢ e₁ ⊑ e₁, (π, II) ⊢ e₁ ⊑ e₁ |
| t var = se₁; se₂| i var = e₁; e₂  | if (π, II) ⊢ se₁ ⊑ e₁, π' = π, II (var = t, l), II' = II, II (var = t, l), (π', II') ⊢ se₁ ⊑ e₂ |
| refining spec| refining spec   | if (π, II) ⊢ se ⊑ e |
| spec            | spec            | if (π, II) ⊢ se ⊑ e |
| invoke(se)      | invoke(e)       | if (π, II) ⊢ se ⊑ e |
| announce p(se)  | announce p(e)   | if (π, II) ⊢ se ⊑ e, (π, II) ⊢ se ⊑ e |
| either {se₁} or {se₂} | if(ep)(e₁) | (π, II) ⊢ se₁ ⊑ e₁, (π, II) ⊢ se₁ ⊑ e₁ |
| either {se₁} or {se₂} | else(e₁) | (π, II) ⊢ se₁ ⊑ e₁, (π, II) ⊢ se₁ ⊑ e₁ |
| either {se₁} or {se₂} | e₁               | (π, II) ⊢ se₁ ⊑ e₁, (π, II) ⊢ se₁ ⊑ e₁ |
| either {se₁} or {se₂} | e₁               | (π, II) ⊢ se₁ ⊑ e₁, (π, II) ⊢ se₁ ⊑ e₁ |

Figure 8: Structural refinement relation ( ⊑ e )

3.2.1 Example Refinement

To illustrate the refinement rules in Figure 8, consider checking whether the handler method update on lines 23–26 in Figure 4 refines the translucid contract’s body on lines 15–16. As illustrated in Figure 9 and according to the rule for se₁; se₂ in Figure 8, this refinement holds if (a) invoke(next) is refined by invoke(rest) and (b) establishes fe==old(fe) is refined by refining establishes fe==old(fe) (Display.update(fe); fe).

For proving condition (a), we must check whether the subexpression next is refined by the subexpression rest. This can be done by the rule for var, which states that both variables next and rest must be given the same type by their respective typing contexts (π and II). The specification typing context π in this case, gives type thunk Fig to next, which is the same as the type for rest given by the program typing context II. By applying the rule for spec in Figure 8, we can prove (b) because specification predicates refining establishes fe==old(fe) are the same in both specification expression and the program expression. Thus, the handler method update correctly refines the translucid contract for event type Changed.

The refinement rule for case spec deserves further explanation. It states that a specification expression spec is refined by an expression refining spec (e), which claims to refine the same specification spec. The claim that e satisfies spec is discharged using runtime assertion checking as discussed in Section 3.4. The rules in Ptolemy’s operational semantics which discharge this condition are shown in Figure 27, rule (Refining).

3.3 Verifying Ptolemy Programs

As previously mentioned, verifying Ptolemy programs is different from verifying standard object-oriented (OO) programs in two ways. First, a method in the program under verification may announce events that can cause a set of handlers to run. Second, if the method is a handler it may call invoke that can also cause a set of handlers to run.

The main difficulty is that the set of handlers (advice) that run in both cases is not known statically unless a whole program analysis is performed. Furthermore, open world assumption of the underlying OO languages precludes us from making sound assumptions about the program’s runtime configuration statically. Thus we must be able to do verification without knowing the program’s entire runtime configuration. Translucid contracts make this possible.

The basic idea is to take the translucid contract of the event type in place of each handler as discussed in Section 3.1. Since from the refinement rules discussed in Section 3.2 each handler will satisfy the contract of the event type, by doing so, we get a sound upper bound on the control effects of the handlers that is independent of the runtime configuration.

3.3.1 Verification of Regular Methods

To statically verify a non-handler method t m ((t m) e) we must replace any occurrence of announce expression in its body e with a simulating expression for verification. The translation function Tr given in Figure 10 shows how to do that. Basically translation function Tr(e, b₁, b₂) inlines event type specification/event body in place of announce/invoke expressions in se, as informally discussed in Section 3.1, to compute a simulating specification expression. Event p is the announced event, if any, and b₁ is the event body. Function Tr is discussed in greater detail in Section 3.3.3.

For the method m above with the body of e we compute Tr(e, skip, λ). The second argument skip and the third argument λ specify that this method is not a handler, thus there is no corresponding event body to run (skip). In other words, non-
3.3.2 Verification of Handler Methods

To statically verify a handler method $m$ handles event $e$, i.e. no event, with the body $\text{skip}$, i.e. no body. These parameters are included in this case simply to facilitate uniform application of the function $T_r$ for both regular methods and handler methods.

The result of $T_r(e, \text{skip}, \bot)$ is a specification expression with no Ptolemy-specific features, but with extra expressions which simulate running of handlers. This expression can be used to perform standard weakest precondition based verification for OO programs.

<table>
<thead>
<tr>
<th>Cases of $p$</th>
<th>Result</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{n. var.}$</td>
<td>var $\perp$</td>
<td>$p = \perp$</td>
</tr>
<tr>
<td>$\text{var (seq)}$</td>
<td>var $\perp$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{if (seq)}$</td>
<td>$\text{if (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{else (seq)}$</td>
<td>$\text{else (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{if (seq) or (seq)}$</td>
<td>$\text{if (seq) or (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{if (seq) and (seq)}$</td>
<td>$\text{if (seq) and (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{either (seq)}$</td>
<td>$\text{either (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{or (seq)}$</td>
<td>$\text{or (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{require (seq)}$</td>
<td>$\text{require (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{assert (seq)}$</td>
<td>$\text{assert (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{do (seq)}$</td>
<td>$\text{do (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{register (seq)}$</td>
<td>$\text{register (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{invoke (seq)}$</td>
<td>$\text{invoke (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
<tr>
<td>$\text{announce (seq)}$</td>
<td>$\text{announce (seq)}$</td>
<td>$\exists p' :: p' = \perp$</td>
</tr>
</tbody>
</table>

Figure 10: Translation algorithm. Algorithm for converting program expressions into specification expressions that simulate running of handlers.

3.3.3 Translation Function

As exemplified in Section 3.1, translation function $T_r(se, b, p)$, with $p$ as announced event and $b$ as the body of $p$, inlines event type specification/event body in place of announce/invite expressions in $se$, to compute a simulating specification. Announce expression is replaced by event type contract whereas invite expression is replaced by either event type contract or event body. Involve expression is replaced by event type’s contract if there are more applicable handlers and is replaced by event body if there is no more applicable handler. As existence of more applicable handlers is not decidable statically, the translation algorithm considers both of the situations in the result either or expression, as shown in Figure 10 in translation of invite and announce expressions.

Most cases of the function $T_r$ are straightforward as they recursively apply $T_r$ to their subexpressions and compose the results. The result of translating refining $\text{spec (e)}$ is $\text{spec}$ as runtime assertion checking ensures that $e$ refines $\text{spec}$. The case of $\text{invoke}$ and $\text{announce}$ expressions is central as they simulate running of handlers. In the translation of these expressions specially for $\text{invoke}$ expression, the assumption is that the contract for event type $p$ is of the form $\text{requires (seq) \text{assures (seq)} \text{ensures (seq)}}$. Consequently in the translation of announce expression the contract for event $p'$ will be like $\text{requires (seq) \text{assures (seq)} \text{ensures (seq)}}$.

Both cases of the translation of $\text{invoke}$ and $\text{announce}$ expressions produce an equivalent choice expression guarded by a refining expression. In case of $\text{invoke (seq)}$, the $\text{either}$ branch contains a sequence of two expressions: result of translating $\text{invoke}$ expression argument of $se_1$ and the event body $b$. The $\text{either}$ branch simulates the situation when there is no applicable handler. The $\text{or}$ branch contains a sequence of two expressions too: result of translating argument $se_1$ and the translation of translucid contract. This simulates the semantics of $\text{invoke}$ expression when there are more applicable handlers. The guarding refining expression assures that specification $\text{spec}$ is satisfied by the choice expression inside.

In case of $\text{announce (p')} (\text{seq})$, the $\text{either}$ branch contains a sequence of two expressions: result of translating announce expression argument of $\text{seq}$ and the translation of event body $se_1$. The $\text{either}$ branch simulates a situation when there are no applicable handlers for event $p'$. The $\text{or}$ branch is most interesting as it simulates announcement of the event $p'$. The first expression in this branch is the result of translating argument expressions and assign them to context variables $\text{seq'}$. The second expression is the translation of the body of the translucid contract for the event $p'$, i.e. $se_1$, assuming that the event body is $se_1$, the translation of $\text{seq}$. The translation of $se_1$ simulates running of handlers for event $p'$ with a concrete event body and event type’s translucid contract as an abstraction for handlers. The $\text{or}$ branch simulates the case when there are some applicable handlers for event $p'$.

The translation function $T_r(e, b, p)$ treats $e$ as a subset of $se \cup \{\text{spec}\}$. Since the syntactic set $se \cup \{\text{spec}\}$ is a strict superset of syntactic set $e$, for every expression $e$ there is an equivalent expression in the set $se \cup \{\text{spec}\}$. The translation function also assumes an acyclic event announce/handle relation. Circular relations could simply be detected statically.

3.3.4 Illustration of Verification Algorithms

To illustrate, consider verifying the method $\text{setX}$ in Figure 4. The body of this method is the announce expression $\text{announce (Changem (this)) (this=X)}$. To verify this method, we first apply the translation function to this ex-
pression with parameters of event body \texttt{skip}, and the event type \texttt{null} as this method is a non-handler regular method. The case for announce expression in Figure 10 is applicable, which results in the specification expression shown in Figure 11.

```
1 refining requires fe != null ensures fe != null
2 either { this ; this.x = x; this } 
3 or { Fig fe = this ; 
4      // ...           this.x = x; this, Changed) }
8      }
9 } establishes fe == old(fe) }
10 }
```

Figure 11: Translation of method \texttt{setX()}

Note the use of translation function on lines 4–5. To verify this expression both the either branch and the or branch must be verified. During this verification, upon reaching the translation function, it is unrolled one more time resulting in the specification expression shown in Figure 12.

```
1 refining requires fe != null ensures fe != null{
2 either { this ; this.x = x; this } 
3 or { Fig fe = this ; 
4      req refining requires fe != null ensures fe != null{
5 either { next; this.x = x; this } 
6 or { next; Tr(invoke(next); establishes fe == old(fe),
7       this.x = x; this, Changed) } 
8 establishes fe == old(fe) }
9 }
```

Figure 12: Unrolling translation function

During this application, the case for sequence expression, the case for \texttt{spec} expression, and the case for \texttt{invoke} expression is used, which also results in an embedded translation function (on lines 6–7). The astute readers may have observed that we have essentially reduced problem of verifying \texttt{announce} and \texttt{invoke} expressions to a problem similar to reasoning about loops. Thus, standard techniques for reasoning about loops can be applied here. Verification of the method \texttt{update} is similar.

### 3.4 Runtime Assertion Checking (RAC)

As previously mentioned, some of the verification obligations encountered during the verification are discharged by relying on runtime assertions. Runtime checking discharges the following obligations, verifying that: (1) a handler method satisfies the relevant event type specification(s) (2) the event body satisfies the pre- and postconditions of its event type specification, (3) each \texttt{refining} expression refines the specification it claims to refine, and finally (4) event announcement and execution of the event handler methods satisfy the pre- and postconditions of the event type specification, regardless of the number of handler and their order of execution.

We have implemented runtime assertion checking in the Ptolemy compiler [16]. Figure 13 illustrates insertion of runtime probes by Polemy compiler in the generated code to guarantee the above-mentioned verification obligations. To meet obligation (1) pre- and postcondition probes are inserted at the beginning and end of handler method body, before line 22 and after line 27. Runtime probes right before and after line 6 guarantee obligation (2). To verify that the refining expression on lines 24-26 refines the specification it claims to refine, obligation (3), runtime assertions are inserted before line 24 and after line 26. Finally to assure obligation (4) that event announcement and execution of handler methods does not violate the event type pre- and postconditions, runtime checks are enforced before and after \texttt{announce} and \texttt{invoke} expressions in the code. Runtime probes before line 5 and after line 7 guarantee the obligation for announce expression whereas probes right before and after line 23 meet the obligations for invoke expressions. In summary, translucid contracts allow modular and independent reasoning about code that announces events and handlers.

### 4. 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 \textit{et al.} [10]. Rinard \textit{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 interactions. Direct interference is concerned about calls to \texttt{invoke}, which is the Ptolemy’s equivalent of AspectJ’s \texttt{proceed}. Direct interference is further categorized into 4 classes of: augmentation, narrowing, replacement and combination advice which call \texttt{invoke} exactly once, at most once, zero and any number of times, respectively. An example, built upon the drawing editor example in Section 1, is shown for each category of direct interference.

#### 4.1 Direct Interference: Augmentation

Informally an augmentation handler evaluates \texttt{invoke} expression exactly once. An augmentation handler can be a before or after handler. In after augmentation, handler is executed after the event body whereas in before augmentation the order is opposite.

```
1 Fig event Changed{
2 Fig fe;
3 requires fe != null
4 assumes {
5 invoke(next); 
6 establishes fe == old(fe)
7 } 
8 ensures fe != null
9 }
```

Figure 14: Specifying augmentation with a translucid contract

To illustrate consider the translucid contract in Figure 14 on lines 3–8. Translucid contracts are required to reveal all appearances of invoke expression, thus it is assured that all refining handlers will evaluate invoke expression exactly once.

Furthermore, \texttt{invoke} is called at the beginning of the contract, requiring event handlers to run after the event body which means that not only the refining handlers are augmentation handlers, but also that they run after the event body, after-augmentation handlers.

Method \texttt{log} in class \texttt{Logging} in Figure 15 is an example of conforming after-augmentation handler. The requirement for this
4.2 Direct Interference: Narrowing

A narrowing handler evaluates \texttt{invoke} at most once, which implies existence of a conditional statement guarding \texttt{invoke}.

To illustrate consider the translucid contract in Figure 16 on lines 5–8 which specifies narrowing handlers. The contract reveals appearances of \texttt{invoke} expression and the \texttt{if} expression guarding that which in turn ensures that invoke expression is evaluated at most once. It does not, however, reveal the actual code of the narrowing handler as long as the hidden code refines the specification on line 8. All refining handlers of the contract will have the same structure in their implementation with regard to \texttt{invoke} and if expressions which guarantees that they are narrowing handlers.

Figure 17 illustrates a narrowing handler refining the contract shown in Figure 16. The handler implements an additional requirement for the figure editor example that “some figures are fixed and thus they may not be changed or moved”. To implement the constraint the field \texttt{fixed} is added to the class \texttt{Fig}, line 23. For fixed figures the value of this field will be 1 and 0 otherwise. The class \texttt{Point} is the same as in Figure 4. To implement the constraint the handler \texttt{Enforce} skips invoking the base code whenever the figure is fixed (checked by accessing the field \texttt{fixed}).

At most one \texttt{invoke}
For a handler to refine the contract in event type Changed, the implementation of the handler Enforce must structurally match the contract. The true block of the if expression on line 14–15 refines the true block of the if expression in the specification on lines 5–6 because as they match textually. The false block of the if expression on line 16–19 refines the false block of the if expression in the specification on lines 7–8 because lines 17–19 claim to refine the specification on line 8. This claim is checked in Ptolemy’s semantics and by runtime assertions as discussed in Section 3.4.

4.3 Direct Interference: Replacement

A replacement handler omits the execution of the original event body and runs the handler body instead. 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 18: Specifying replacement with a translucid contract

Figure 18 shows the event type Moved that its contract specifies augmentation handlers by not having any invoke expression in the contract, line 6.

```java
class Scale{
  int s;
  Fig scaleit(thunk Fig rest, Point p, int d){
    refining preserves p!=null && ...
    Fig{
      int x, int y;
      Fig moveX( int d){
        announce Moved(this , d){
          this .x += d; this
        }
      }
    }
  }
}
```

Figure 19: Replacement handler

Figure 19 shows an example of a replacement 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 scaling factor” defined in class Scale”.

Translucid contracts are obliged to reveal all appearances of invoke expressions. Thus if an event type’s contract has no invoke expression, none of the event type’s handlers are allowed to have an invoke in their implementation. Otherwise the structural similarity criterion of refinement is violated. The handler scaleit refines Moved’s contract because its body on lines 13–15 matches the specification. There is no invoke expression and the invariant expected by the event type’s contract on line 6 and that maintained by the body on line 14 are the same.

4.4 Direct Interference: Combination

Combination handlers can evaluate 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.

Figure 20: Combination contract and handler

We show an example of a combination contract and handler in Figure 20. The translucid contract in the event type specification on lines 5–11 allows an invoke expression to be evaluated zero or more number of times. This is achieved by revealing evaluation of invoke expression guarded by while expression. By analyzing the specification, specially having while loop revealed, the base code developer can conclude that event handler methods of ClChange may run the original event body multiple times. The developer, however, is not aware of the concrete details of event handlers, thus those details remain hidden.

A combination handler is illustrated in Figure 20 lines 15–34. In this example, figures in the drawing editor are extended 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, lines 35–43. 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 it, 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 39–41. The body of the announce expression contains the code to obtain the next color on line 40. The handler Unique on lines 15–34 implements this requirement by storing already used colors in a hash table (colors). The field colFix is added to class Fig to show that a unique color has been chosen and fixed for the figure. When the handler method check is run it checks colFix to see if a color has been chosen.
yet or not. If not it then invokes the event body generating the next candidate color. If the 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 colFix is set to 1, lines 21–26.

The specification for the event type ClChange on lines 4–12 documents that a combination handler will be run when this event is announced. This specification makes use of the choice feature, on line 7–10. To correctly refine this specification, according to Figure 8 refinement rules, 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. Refinement is illustrated in the figure showing which specification block is refined by which block of the implementation.

4.5 More Expressive Control Flow Properties

Rinard et al.’s control flow properties are only concerned about calls to invoke. Their proposed technique decides which class of interference and category of control effects each isolated advice belongs to [10]. However, it can not be used to analyze the possibility of two or more control flow paths each of which an augmentation, if each path maintains a different invariant. Figure 21 illustrates such a scenario with an example adapted from [7].

![Figure 21: Expressive control flow properties beyond [10]](image)

The class Fig not shown here is the same as in Figure 4. Khatchadourian and Soundararajan [7] implement an additional requirement that “a point should be visibly distinguished from the origin”. To implement this requirement a scaling factor s is added to the class Point as a field member on line 29, initially set to 1 on line 32. The requirement is implemented in the class Scaling. The handler method scaleIt is run whenever event Moved is announced. The handler ensures that if the point is close enough to the origin (vicinity condition), to visibly distinguish it from the origin by setting the scaling factor to 10. The scaling factor only has two values: 1 and 10. The vicinity condition is true if the point’s x and y coordinates are both less than 5.

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 is set to 10 if and only if the vicinity condition holds. Rinard et al.’s proposal could only be used to verify (i) and a behavioral contract could specify (ii) but none of them could specify (iii), whereas our approach can. On lines 6–9 there is a specification that conveys to the developer of the class Point that a conforming handler method will satisfy all of the three above-mentioned assertions.

In summary, in this section we have shown that translucid contracts allow us to specify control flow interference between a subject and its observers. Specified interference patterns are enforced automatically through structural 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 specified by previous works on design by contract for aspects, which could be specified as translucid contracts.

5. APPLICATIONABILITY TO OTHER AO IN-TERFACES

We now 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) [1], Sullivan et al.’s crosscutting interfaces (XPis) [2], Aldrich’s Open Modules [3], and Steimann et al.’s join point types [5]. We have tried out several of these ideas and our approach works beautifully. Since Steimann et al.’s join point types [5] and Hoffman and Eugster’s explicit join points (EJP) are similar in spirit to Ptolemy, which we have already discussed in previous sections, we do not present the straightforward adaptation of our ideas to their work here. Rather we focus on the AspectJ implementation of the XPI approach [2], Kiczales and Mezini’s AAI [1], and Aldrich’s Open Modules [3] that are substantially distinct from event types [6, Fig. 10].

5.1 Translucid Contracts for XPis and AAI

Sullivan et al. [2] 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 join points, how they are exposed through the join point model of the given language, and constraints on behavior across join points (e.g. provides and requires conditions [2]). XPis prescribe rules for join point exposure, but do not provide a compliance mechanism. Sullivan et al. have shown that at least some design rules can be enforced automatically using AspectJ’s features [2]. Current proposals for XPis, however, all use behavioral contracts [2]. As shown previously, use of behavioral contracts, limits the expressiveness of the assertions which could be made using XPI. Behavioral contracts cannot reveal control flow details, which might be needed for reasoning about interference from control effects in cases such as those discussed above.

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 are needed to verify such programs. To illustrate, consider the narrowing example from Section 4.2 shown in Figure 22 and Figure 23, where the constraint on movement of figures is implemented as an XPI and an aspect. Figure 22 shows the
XPIs typically also contain a description of which scope, 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 7. Other than that, our syntax works right out-of-the-box.

Our proposal for verifying refinement also needs only minor changes. Figure 23 shows a refining advice for the XPI contract of Figure 22. Unlike Ptolemy, where the event types of interest are specified in the binding declarations, in Sullivan et al.’s version of XPIs, aspects reuse the pointcut declarations from the XPI in the advice declaration (lines 14). 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, 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 (pcd1 && pcd2 or pcd1 || pcd2), where both pcd1 and pcd2 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.

Join point interfaces like XPIs could be computed from the implementation rather than being explicitly specified, given whole-program information. Kiczales and Mezini [1] 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 [2]. However, an important property of AAI is that advised join points contain the details of the advice. An example based on the narrowing example of Section 4.2 is shown in Figure 24. 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 join point shadow as shown on lines 5–12.

Unlike translucid contracts for event types in Ptolemy, where the contract is thought of as attached to the type, in the XPI, contracts are thought of as attached to the pointcut declaration, e.g. the contract on lines 4–11 is attached to the pointcut on lines 2–3. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract can only use fe.

Figure 25 shows the implementation of the same scenario using Open Modules. In implementing the example, we use the syntax from the work of Ongkingso et al. [20] to retain similarity.
with other examples. In the listing constraints on the movement of figure is encapsulated in the module (aspect) Enforce in Figure 26. Open module FigModule in Figure 25 exposes a pointcut of class Fig on line 2–3, marked by the keyword expose to. The exposed pointcut is advisable only by the aspect Enforce. The translucid contract on lines 4–11 states the behavior of interaction between specified aspect Enforce as shown in Figure 26 and the exposed pointcut through expose to construct. The adaptations in the syntax of contracts are the same as in the case of the XPIs discussed in Section 5.1.

Like contracts in XPIs, contracts in Open Modules are attached to a pointcut declaration, e.g. the contract on lines 4–11 is attached to the exposed pointcut defined on lines 2–3. The variables that can be named in the contract are those exposed by the pointcut. For example, the contract can only use the variable fe.

The rules proposed for verifying refinement 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 14–15). Our refinement rules could be added here in an AO type system. Well-formedness of basic and compound pointcuts follow the same rules laid out in Section 5.1.

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.

6. 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 specification.

6.1 Contracts for Aspects

This work is closest in the spirit to the work on crosscutting programming interfaces (XPIs) [2]. 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 clause specifies obligations of the code that handles events. However, the contracts specified by these works are mostly informal behavioral contracts and can not be automatically checked. Furthermore, these works do not describe a verification technique and contracts could be bypassed.

Skotiniotis and Lorenz [21] propose contracts for both objects and aspects in their tool Cona. Cona’s contracts are black box, and thus do not reveal any information about control flow effects.

Similarly, Pipa is a behavioral specification language for AspectJ [8]. 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. AspectJ program annotated with JML-like Pipa’s specifications could be transformed into JML and Java code. JML-based verification tools could enforce specified behavioral constraints. All of these ideas use behavioral contracts and thus may not be used to reason about control effects of advice.

6.2 Modular Reasoning

There is a large body of work on modular reasoning about AO programs on language designs [3,4,17], design methods [1,2], and verification techniques [22,23]. Our work complements ideas in the first and the second categories and can use ideas in the third category for improved expressiveness. Compared to work on reasoning about implicit invocation [24,25], our approach based on structural refinement is significantly lightweight. Furthermore, it accounts for quantification that these ideas do not.

Oliveira et al. [26] 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 interference 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’s explicit join points [17] and Steimann et al.’s join point types [5] share similar spirit with Rajan and Leav’s event types [6]. Although Steimann et al. proposed informal behavioral specification, but there is no explicit notion of formally expressed and enforced contracts, or stating interaction behavior, in any of these approaches.

The work of Khatchadourian et al. [27] is closely related in that it addresses both specification and modular verification of AO programs. They use a rely-guarantee approach to specification and verification. Black box behavioral specifications are attached to PCDs in pointcut interfaces, in a way similar to our work. The assumes part of a translucid contract plays a role similar to the rely conditions in their specifications, since it specifies the possible state transformations that advice may implement. Structural refinement in our approach plays a role similar to the guarantee part of their specification, since it also limits what the advice (or handler) can do. The main difference is that they use “join point traces” to reason about control effects, which adds an extra burden on the specifier and verifier compared to our grey box approach, which allows more traditional reasoning about control effects in terms of the underlying programming language’s control flow. Their approach is based on black box behavioral specification.

6.3 Grey Box Specification and Verification

This work builds upon previous research on grey box specification and verification [11]. Among others, Barnett and Schulte have considered using grey box specifications written in AsmL [28] for verifying contracts for .NET framework, Wasserman and Blum [29] also use a restricted form of grey box specifications for verification, Tyler and Soundarajan [30] and most recently Shaner et al. [15] have used grey box specifications for verifica-
tion of methods that make mandatory calls to other dynamically-dispatched methods. Rajan et al. have used grey box specification to enable expressive assertions about web-services [18]. Compared to these ideas, our work is the first to consider grey box specification as a mechanism to enable modular reasoning about code that announces events and handles events, which is a common idiom of AO and II languages.

7. DISCUSSION

The benefits of assertion checking come with a price. The price to be paid is the additional annotations which should be added in the code to enforce assertions. Our proposal is not an exception to this rule. The amount of annotations needed for the event type, advising code and the advised code is a key factor that determines the practicality of our proposal.

On the advised code side, an event could be announced in many places. Frequent event announcement along with annotations required at event announcement sites motivate the need to limit the amount of annotations needed at the event announcement sites. To that end, a benefit of our proposal is that it does not require any annotations at the event announcement site. On the advising code, many handler methods may run when an event is announced. Thus, the annotation overhead for the handler methods should also be fairly small. We only require the use of the **refining** expressions where the specification hides the handler method’s implementation details. Thus, the annotation burden for the handler methods is also fairly small. Furthermore, since the structure of the handler methods must be similar to the translucid contract, and the translucid contract uses specification expressions to hide the code for the handler method, for most cases it is easy to infer the specification for the hidden code from the translucid contract.

8. CONCLUSION AND FUTURE WORK

This paper has shown how to modularly specify and verify Ptolemy programs that use dynamically announced events and handlers, which is similar to AspectJ’s pointcuts and dynamic advice. There are several key ideas involved in our solution.

First, using Ptolemy [6] provides a notion of event type declarations. Event announcement names an event type, and so code announcing an event can use the translucid contracts given in the event type declaration. Similarly, handlers are statically bound to event types in binding declarations, and this allows binding verification to also modularly refer to the event type’s translucid contract. As the interface between event announcements and handlers, event type declarations are thus a good place to write translucid contracts. We also demonstrate the applicability of our techniques to other type of AO interfaces [1–3, 5, 17] in our technical report [31].

Finally, and most importantly, using grey box specifications as part of our translucid contracts, and using structural refinement in verification solves the problem of reasoning about control effects of handlers. In essence, the grey box specification exposes all the interesting control effects of handlers and structural refinement ensures that correct handler implementations are limited to the specified control effects. We argued that black box behavioral contracts are insufficient for reasoning about such control flow effects, but showed how our translucid specifications were adequate to specify a wide variety of such control effects.

We have added translucid contracts to a Ptolemy compiler that verifies handler refinement and inserts runtime assertion checking code [16]. Adding translucid contracts to other AO compilers, integrating our ideas with the rich specification features of JML, and working out larger examples are some directions for future work.

Acknowledgments

Rajan and Bagherzadeh were supported in part by the NSF grant CNS-06-27354 and CCF-08-46059. Leavens and Rajan were supported in part by NSF grant CNS-08-08913.

APPENDIX

A. SOUNDNESS OF REASONING

To reason about a method’s body (e) containing announce and invoke expressions, we use the translation algorithm shown in Figure 10 to generate a simulating specification expression (se) (see Section 3). We claim that the method body expression e is a Hoare logic-based refinement of generated simulating specification expression se [15]. In other words, if starting with a precondition state φp, the specification expression se implies the postcondition state φq, then starting with the same precondition state φp, and by running e, we will reach the postcondition state φq. This condition is formalized in the definition below.

**Definition 1. (Hoare Logic Refinement)** A specification expression se is said to be Hoare-logic-refined by expression e, expressed as se ⊑ e, if and only if for all predicates over program states φp and φq, φp(se) ⇒ φq(se) ⇒ φq(e).

To prove our claim, we rely on Shaner et al.’s work on reasoning about object-oriented programs that contain specification expressions [15]. This work proves that an object-oriented program expression eoo is a Hoare-logic refinement of an object-oriented specification expression sseoo, if eoo’s structure matches sseoo’s structure and for every specification expression sspec in sseoo, there is a corresponding refining expression in eoo, that claims, and is verified to, refine sspec according to Hoare logic. We incorporate their result as the lemma below.

**Lemma 1. (Shaner-Leavens-Naumann Soundness)** Let sseoo and eoo be specification and program expressions and let sseoo ≡ eoo, as defined in Figure 8, then for all predicates over program states φp and φq, φp(seoo) ⇒ φq(eoo) ⇒ φq(e).

But Shaner et al. only prove their results for object-oriented expressions (meaning the expressions in their paper [15]). To apply these results to reasoning about Ptolemy programs, we must reduce both Ptolemy-specific specification expressions and program expressions to object-oriented expressions (from [15]). Below we give some sub-results along those lines.

**Lemma 2. (Translation Produces Object-Oriented Specification Expressions)** Let ssept be an expression which may contain Ptolemy-specific expressions and let sseoo be the result of the application of applying the translation algorithm shown in Figure 10 to ssept, i.e. sseoo = Tr(sept, skip, ⊥). Then sseoo is an object-oriented specification expression.
A.1 Substitution Algorithm

The substitution and translation algorithms are similar on one hand, in the sense that they both replace announce and invoke expressions, on the other hand, they are different as substitution algorithm produces a program expression by replacing announce and invoke expressions, whereas translation algorithm results in a specific expression. The translation algorithm replaces announce and invoke expressions with either the event type’s contract or the event body, depending on the existence of applicable handlers. The substitution algorithm replaces those expressions with either body of the next handler or event body, again based on the existence of applicable handlers. \( \text{Subst}(e, b, p, \text{loc}_A) \) is the application of substitution algorithm to program expression \( e \), with event \( p \) announced and event body \( b \). Instead of list of active objects \( A \) in Ptolemy’s original semantics, the substitution algorithm uses a constant memory location \( \text{loc}_A \). Location \( \text{loc}_A \) points to an object of class \text{ActiveList} \, which is responsible for tracking the list of receiver objects for applicable handlers.

Most cases of substitution algorithm \( \text{Subst} \) are straightforward: like those of the translation algorithm, they recursively apply \( \text{Subst} \) to each subexpression and compose the results. Figure 29 shows how to do that. For Ptolemy-specific expressions, the rule for \text{refining} spec\{e\} basically applies the substitution algorithm to the subexpression \( e \). The rule for \text{register}(e), first applies the substitution algorithm to the subexpression \( e \) and then adds it to the list of the applicable handlers. The most interesting cases are those for the invoke and announce expressions. In the substitution of these expressions specially for invoke expression, the assumption is that the contract for event type \( p \) is of the form \text{requires} sp_p \text{ assumes} \{ \text{se}_p \} \text{ ensures} sp_p. Consequently in the substitution of announce expression the contract for event \( p \) will be like \text{requires} sp_p \text{ assumes} \{ \text{se}_p \} \text{ ensures} sp_p'.

In both cases conditional if expressions are produced as the body of a refining expression. The refining expression claims to refine the black box behavioral specification spec of the event type \( p \). The refinement of the specification expression by the body of a refining expression is taken care of by run time assertion checking, as discussed in Section 3.4.

\( \text{Subst}(\text{invoke}(e), b, p, \text{loc}_A) \) produces a conditional if expression which checks for the number of applicable handlers. In its true branch, the conditional expression, contains a sequence of two expressions: substitution of parameter expression \( e \) and substitution of the event body \( b \), with the assumption that there are no more applicable handlers. Likewise, the false branch of the conditional contains a sequence of two expressions: result of the substitution of parameter expression \( e \) and result of the substitution of the body of the next applicable handler. The assumption of this branch is the existence of more applicable handlers. Compare this to the translation of invoke expression in Section 3.3.3.

In case of an announce expression announce \( p' \ (e) \ [e] \), the result of substitution is again a conditional if expression checking for the number of applicable handlers. The true branch of the conditional contains a sequence of two expressions: substitution of parameter expression \( e \) and substitution of the event body \( e \). The assumption in this branch is that there are no more applicable handlers. The false branch of the conditional contains a sequence of two expressions as well: result of the substitution of parameter expression \( e \) and result of the substitution of the body of the next applicable handler. Readers are encouraged to compare this to the translation of announce expression in Section 3.3.3.

Figure 30 shows auxiliary functions used in the substitution algorithm. Function \( \text{suc}(\text{loc}_A, p) \) returns the body of the next handler of event \( p \) using the location \( \text{loc}_A \), which points to the list of

\[
\begin{align*}
(\text{REGISTER}) & \quad e' = \text{Subst}(\text{register}(e), \text{skip}, \ell, \text{null}) \\
& \quad \text{register}(e), J, S \mapsto (e', J, S) \\
(\text{ANNOUNCE}) & \quad e' = \text{Subst}(\text{announce} p (e) \ [e], \text{skip}, \ell, \text{null}) \\
& \quad \text{announce} p (e), J, S \mapsto (e', J, S) \\
(\text{REFINING}) & \quad n \neq 0 \\
& \quad (\exists e' \text{refining requires } n \text{ ensures } e'[e]) \mapsto (\exists e' \text{evalbody } e'), J, S) \\
(\text{EVALBODY}) & \quad \rho = \text{envOf}(\nu) \quad \Pi = \text{tenvOf}(\nu) \\
& \quad t = \Pi(e) \quad \rho' = \Pi[e\{\text{result } : v\}] \\
& \quad \Pi' = \Pi[e\{\text{result } : \text{var } t\}] \quad \nu' = \text{lexframe} \rho' \Pi' \\
& \quad (\exists \text{evalbody } \nu e), J, S \mapsto (\exists \text{under evalpost } \nu e), \nu' + J, S) \\
(\text{EVALPOST}) & \quad n \neq 0 \\
& \quad (\exists \text{under } \nu e), J, S \mapsto (\exists \nu e), J, S)
\end{align*}
\]
applicable handlers for event \( p \). The function gets the location of the first handler of event \( p \) by calling method \( \text{getFirst()} \) and performs a standard \( \| \)-reduction on the handler method’s body. \( \alpha \)-renaming takes care of name clashes, if any. Auxiliary function \( \text{findHandler}(c, p, CT) \) returns the handler for event \( p \) in class \( c \) where \( CT \) is a list of program declarations. Function \( \text{eventsOf}(CT, loc) \) returns a list of events that object \( loc \) observes.

```java
1 class ActiveList {
2  Hashtable hash;
3  LinkedList handlers(Event p){
4   LinkedList hList = null;
5   hList = ...
6    add(o,p)
7   }
8  }
9 }
10 class HashTable {…}
11 class LinkedList {…}
12 class Event {…}
```

Figure 28: Classes to simulate list of active objects

To implement the substitution algorithm we assume the existence of some pre-defined classes like \( \text{ActiveList} \) as shown in Figure 28. \( \text{ActiveList} \) keeps track of the list of active objects per event type. Handlers of each specific event are stored in a \( \text{LinkedList} \). Constant location \( \text{locA} \) points to an object of type \( \text{ActiveList} \). Method \( \text{add(Object o, LinkedList evs)} \) in \( \text{ActiveList} \) adds object \( o \) as the observer for all the events in the list \( evs \). Classes \( \text{HashTable} \) and \( \text{Linkedlist} \) are the same as classes \( \text{HashTable} \) and \( \text{Linkedlist} \) in Java. \( \text{ActiveList} \) has an extra method \( \text{tail} \) which returns the tail of the list.

A.2 Proof of Soundness

To prove the soundness of our reasoning approach, we should prove the translation algorithm sound, i.e., that the specification expression produced by translation algorithm used for reasoning is refined by the program expression produced by substitution algorithm. Theorem 1 formalizes this.

To reason about a method which may announce an event, translation algorithm is applied to the method body, \( \delta p t \), which may include Ptolemy-specific expressions and the result specification expression \( \text{spec}_\text{oo} \) is used to reason about the method. Lemma 2 assures \( \text{spec}_\text{oo} \) is an OO specification expression and therefore can be used for reasoning purposes based on Shaner et al.’s approach [15] as stated by Lemma 1. This is possible only if there is a guarantee that \( \text{spec}_\text{oo} \) is specifying the runtime behavior of the method. The sub-

### Figure 28: Classes to simulate list of active objects

#### Figure 29: Substitution algorithm

\[
\text{spec}_\text{oo} = \begin{cases}
\delta & \text{if } \text{loc} \neq \text{null}, \\
\text{null} & \text{otherwise}
\end{cases}
\]

When \( \text{loc} \) is \( \text{null} \), \( \text{spec}_\text{oo} \) reduces to \( \text{null} \). When \( \text{loc} \) is \( \text{non}-\text{null} \), \( \text{spec}_\text{oo} \) reduces to \( \delta \).

**Theorem 1.** (Refinement Theorem) Let program expression \( e \) be the body of a method and \( \text{spec}_\text{e} = \text{Tr}(e, \text{skip}, \perp) \) be the translation of \( e \). Let \( \text{spec}_\text{e} \) be the substitution of \( e \). Then: \( \text{spec}_\text{e} \preceq \text{spec}_\text{e} \).

**Proof Sketch:**

The proof is by induction on the cases of expression \( e \). For each case we prove \( \text{spec}_\text{e} \preceq \text{spec}_\text{e} \) as defined in Figure 8 and then conclude \( \text{spec}_\text{e} \preceq \text{spec}_\text{e} \) based on Lemma 1 and definition 1. Proof given based on the case of \( e \) where \( e \) is a non-specification
expression. Thus specification expressions next, old(se), either {se} or {se}, requires sp ensures sp are not considered in the proof.

- \( e \in \{n, \text{var}, \text{null}, \text{new } c() \} \), this is vacuously true because \( se' = e \) and \( e' = e \) and any expression is refined by itself, i.e., \( e' \subseteq e \). Therefore \( se' \subseteq e' \) which in turn implies \( se' \succ e' \) based on lemma 1 and definition 1.

- \( e = e.m(e) \), where \( se' = Tr(e.m(e), \text{skip}, \_p) \) and \( e' = \text{Subst}(e, m(e), \text{skip}, \_p, \null) \). Based on the induction hypothesis a subexpression in \( se' \) is refined by its corresponding subexpression in \( e' \). And based on the definition of the translation and substitution algorithms it is easy to see that \( se' \subseteq e' \) and \( e' \) are structurally similar. Therefore \( se' \subseteq e' \).

- For \( e \in \{ e.f, e.f.e = e, \text{if}(e)(e) \} \) else(e), cast e, e; e, while(e)(e), t var = e; e), the proof is similar to the proof for method call case of \( e = e.m(e) \).

- \( e = \text{refining spec} \), where \( se' = \text{spec} \) and \( e' = \text{refining spec} \{ \text{Subst}(e, \text{skip}, \_p, \null) \} \). Refining expression \( e' \) is refining specification expression \( \text{spec} \) which is the same as \( se' \).

- \( e = \text{register}(e) \), based on the induction hypothesis a subexpression in \( se' \) is refined by its corresponding subexpression in \( e' \). As it can be seen the substitution of register expression is manipulating the list of active objects through \( \text{loc}_x \). An Unrolling strategy in the specification expression generated by translation algorithm takes care of different number of handlers.

### Figure 31: Structural similarity of translation and substitution of announce and invoke expressions

- \( e = \text{invoke}(e) \), again induction hypothesis asserts a subexpression in \( se' \) is refined by its corresponding subexpression in \( e' \). Also recall that each handler method refines its event type specification which means refinement of \( sp \) by the body of the next handler \( \text{sub}(\text{loc}_x, p) \). Structural similarity of \( se' \) and \( e' \) could easily be seen in Figure 31. Translation and substitution of invoke and announce expressions is shown in this figure. Refinement rules in Figure 8 assure either-or block on translation side for invoke expression in Figure 31 is refined by if-else block on substitution side.

- \( e = \text{announce } p'(\pi)(\{se\}) \). Based on the induction hypothesis a subexpression in \( se' \) is refined by its counterpart subexpression in \( e' \). Structural similarity of \( se' \) and \( e' \) could easily be seen in Figure 31.


