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

Mehdi Bagherzadeh[β], Hridesh Rajan[β], Gary T. Leavens[θ] and Sean Mooney[β]
[β]Iowa State University, Ames, IA, USA
{mbagherz, hridesh, smooney}@iastate.edu
[θ]University of Central Florida, Orlando, FL, USA
leavens@eecs.ucf.edu

ABSTRACT

As aspect-oriented (AO) programming techniques become more widely used, their use in critical systems such as aircraft and telephone networks, will become more widespread. However, careful reasoning about AO code seems difficult because: (1) advice may apply in too many places, and (2) standard specification techniques do not limit the control effects of advice. Commonly used black box specification techniques cannot easily specify control effects, such as advice that does not proceed to the advised code. In this work we avoid the first problem by using Ptolemy, a language with explicit event announcement. To solve the second problem we give a simple and understandable specification technique, 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 sound modular verification of typical interaction patterns used in AO code. We also show that translucid contracts allow interesting control effects to be specified and enforced.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Programming by contract, Assertion checkers; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions, Invariant, Pre- and post-conditions, Specification techniques

General Terms

Design, Languages, Verification

Keywords

Translucid contracts, modular reasoning, implicit invocation, aspect-oriented interfaces, grey box specification, Ptolemy

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 frequently (e.g., at each method or constructor call and each field read and write). And at each join point shadow, reasoning 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.

1.1 Density of Join Point Shadows

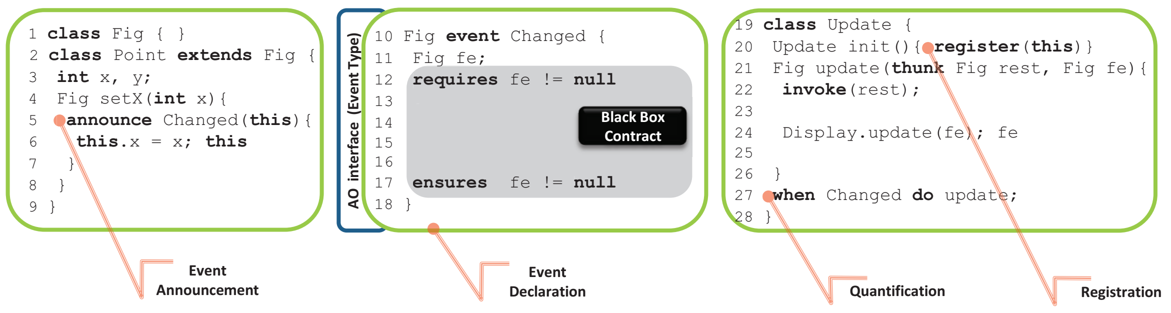
As an example of the first problem, consider the straight-line code in below. In this listing, assuming that x and y are fields, there are at least 8 join point shadows, including the 5 method calls, the writes of x and y, and the read of x.

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

Knowing what advice applies where is amenable to tool support. An example is the Eclipse AspectJ Development Tools (AJDT). The idea of aspect-aware interfaces [14], is equivalent to such tool support. However, the number of reasoning tasks grows with the number of join points and the amount of applicable advice.

One way of avoiding this problem of frequent occurrence of join point shadows, 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, [1,7,19,26,27]. This is the approach we adopt in this paper by using the language Ptolemy [19]. Ptolemy introduces the notion of event types and limits the join points to explicit event announcements.

To illustrate, consider the Ptolemy code in Figure 1 from the canonical drawing editor example with functionalities to draw points, lines and update the display. In Ptolemy, events are explicitly announced, which mitigates the 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 10–18 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 *explicitly* and *declaratively* created using **announce** expressions as shown on lines 5–7. Like Eos [20, 21], Ptolemy doesn't distinguish between aspects and classes. On lines 19–28 is the Ptolemy's equivalent of an AspectJ-like advice, which advises calls to the method setX. The Update class has a binding declaration on line 27



```
1 class Fig { }
2 class Point extends Fig {
3  int x, 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  requires fe != null
13
14
15
16
17  ensures  fe != null
18 }
```

```
19 class Update {
20  Update init(){register(this)}
21  Fig update(thunk Fig rest, Fig fe){
22   invoke(rest);
23
24   Display.update(fe); fe
25
26  }
27  when Changed do update;
28 }
```

Figure 1: A behavioral contract for aspect interfaces using Ptolemy [19] as the implementation language. See Section 2.1 for syntax.

that says to run the handler method `update` whenever events of type `Changed` are signaled. In Ptolemy's terminology advice are called *handlers*. Ptolemy also provides dynamic registration using **`register`**, as shown on line 20, which activates the current instance of the `Update` class as an observer for the event `Changed`.

1.2 Reasoning about Control Effects

As an example of the second problem, understanding control effects of the advice, consider the `Logging` handler in the listing below that advises the same set of events advised by `Update` handler in Figure 1. To understand the control flow at these events matched by these handlers a developer must understand the control flow of both handlers. Furthermore, to understand the behavior at such events one must also understand the control flow of all other handlers that may advise the same events.

```
29 class Logging{
30  …
31  Fig log(thunk Fig rest, Fig fe){
32   invoke(rest);
33   Log.logChanges(fe); fe
34  }
35  when Changed do log;
36 }
```

Design by contract (DBC) methodologies for aspect-oriented software development (AOSD) have been explored before [13, 27, 30], 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. As shown in Figure 1, event type `Changed` declares a black box contract on lines 12–17. Phrases "behavioral contract" and "black box contract" are used interchangeably throughout the paper.

However, the black box contract on lines 12–17 does not specify the control effects of the handler. For example, with just the black box contract of the event type `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). If the expression **`invoke`** in the handler method `update` is forgotten inadvertently, the execution of the body of method `setX` will be skipped. This is equivalent to missing the call to **`proceed`** in an advice in AspectJ. Such assertions are important for reasoning, which depends on understanding the effect of composing the handler modules with the base code [23, 27]. That is, the contract does not specify whether the handler must always proceed.

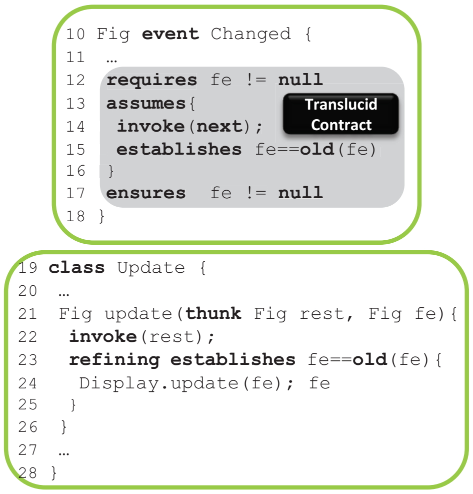
This limitation of black box contracts was discussed in a preliminary version of this paper [2]. Ideas from Zhao and Rinard's Pipa language [30], if applied to AO interfaces help to some extent. However, as we discuss in greater detail in Section 5, Pipa's expressiveness beyond simple control flow properties is limited.

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 [27]. But black box 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.

Another problem with such black box contracts is that they do not help with effectively reasoning about the effects of aspects on each other. Consider another example concern, say `Logging`, which writes a log file at the events specified by `Changed`. For this concern different orders of composition with the `Update` concern in Figure 1 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 22 of Figure 1 was omitted; that is, suppose that `Update` handler 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 handlers were composed in the opposite order. A handler developer cannot, by just looking at the black box contract of the event type, reason about the composition of such handlers. Rather a developer must be aware of the control effects of the code in all composed handlers. Furthermore, if any of these handlers changes (i.e., if their control effects change), one must reason about every other handler that applies at the same events.

The main contribution of this work is the notion of *translucid contracts* for AO interfaces, which is based on grey box specification [6]. 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 [16], but we use a restricted form of refinement that requires structural similarity, to allow specification of control effects.

We have added an example translucid contract to the AO interface, event type `Changed`, on lines 12–17 of Figure 2. Unlike a 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



```
10 Fig event Changed {
11  …
12  requires fe != null
13  assumes{
14   invoke(next);
15   establishes fe==old(fe)
16  }
17  ensures  fe != null
18 }
19 class Update {
20  …
21  Fig update(thunk Fig rest, Fig fe){
22   invoke(rest);
23   refining establishes fe==old(fe){
24    Display.update(fe); fe
25   }
26  }
27  …
28 }
```

Figure 2: A translucid contract for event type `Changed`

occurrence of the **`invoke`** expression (which is like AspectJ's proceed) in the handler method *must* be made explicit in the translucid contract, line 14. 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 line 14 one may conclude that, irrespective of the concrete handler methods, the body for the method `setX` on line 6 of Figure 1 will always be run. Such conclusions allow a 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. Expression **`next`** is a specification placeholder for the event closure passed to the handlers.

Requiring the **`invoke`** expression to be made 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 contracts for describing and reasoning about control flow effects.

To soundly reap these benefits, the translucid contract for the event type `Changed` must be refined by each conforming handler method [16]. We borrow the idea of structural refinement from JML's model programs [24] and enhance it to support AO interfaces, which requires several adaptations that we discuss in Section 3. Briefly the handler method `update` on lines 22–25 in Figure 2 refines the contract on lines 12–17 because line 22 matches line 14 and lines 23–25 claim to refine the specification expression on line 15. The pre- and postconditions of `update` are considered the same as the pre- and postconditions of event type specification on lines 12 and 17, respectively.

1.3 Contributions

In summary, this work makes the following contributions:

- A specification and verification technique for writing contracts for AO interfaces and a proof of the soundness of the presented specification, verification and reasoning approach;
- An implementation of the proposed specification and verification technique in Ptolemy's compiler [18];
- An analysis of the effectiveness of our contracts using Rinard *et al.*'s work [23] on aspect classification which shows our technique works well for specifying all classes of aspects (as well as others that Rinard *et al.* do not classify);
- 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 [19] for this discussion and to adapt Ptolemy's syntax and semantics descriptions. 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 3 and explained below. A Ptolemy program consists of zero or more declarations, and a "main" expression (see Figure 1 and Figure 2). Declarations are either class declarations or event type declarations.

$$\begin{array}{lll}
\textit{prog} & ::= & \overline{\textit{decl}}\ e \\
\textit{decl} & ::= & \mathtt{\textbf{class}}\ c\ \mathtt{\textbf{extends}}\ d\ \{\ \overline{\textit{field}}\ \ \overline{\textit{meth}}\ \ \overline{\textit{binding}}\ \} \\
 & & \mid t\ \mathtt{\textbf{event}}\ p\ \{\ \overline{\textit{form}}\ \ \textit{contract}\ \} \\
\textit{field} & ::= & t\ f\,\mathtt{;} \\
\textit{meth} & ::= & t\ m\ (\overline{\textit{form}})\ \{\ e\ \} \mid t\ m\ (\mathtt{\textbf{thunk}}\ t\ \textit{var},\ \overline{\textit{form}})\ \{\ e\ \} \\
\textit{form} & ::= & t\ \textit{var}, \quad \textbf{where}\ \textit{var} \neq \mathtt{\textbf{this}}\ \text{and}\ \textit{var} \neq \mathtt{\textbf{next}} \\
\textit{binding} & ::= & \mathtt{\textbf{when}}\ p\ \mathtt{\textbf{do}}\ m \\
e & ::= & n \mid \textit{var} \mid \mathtt{\textbf{null}} \mid \mathtt{\textbf{new}}\ c\,() \mid e.m(\,\overline{e}\,) \mid e.f \mid e.f = e \mid \textit{form} = e\mathtt{;}\ e \\
 & & \mid \mathtt{\textbf{if}}\ (\textit{ep})\ \{\ e\ \}\ \mathtt{\textbf{else}}\ \{\ e\ \} \mid \mathtt{\textbf{while}}\ (\textit{ep})\ \{\ e\ \} \mid \mathtt{\textbf{cast}}\ c\ e \mid e\mathtt{;}\ e \\
 & & \mid \mathtt{\textbf{register}}\,(\ e\ ) \mid \mathtt{\textbf{invoke}}\,(\ e\ ) \mid \mathtt{\textbf{announce}}\ p\ (\ \overline{e}\ )\ \{\ e\ \} \\
 & & \mid \mathtt{\textbf{refining}}\ \textit{spec}\ \{\ e\ \} \\
\textit{ep} & ::= & n \mid \textit{var} \mid \textit{ep}.f \mid \textit{ep}\ \mathtt{!=}\ \mathtt{\textbf{null}} \mid \textit{ep} == n \mid \textit{ep} < n \mid\ !\ \textit{ep} \mid \textit{ep}\ \&\&\ \textit{ep}
\end{array}$$

where

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

Figure 3: Ptolemy's syntax [19], with `refining` expressions and contracts added

2.1.1 Declarations

We do not allow nesting of *decl*s. A class has a name (c) and names its superclass (d), and may declare fields ($\overline{\textit{field}}$) and methods ($\overline{\textit{meth}}$). Field declarations are written with a class name, giving the field's type, followed by a field name. Method headers 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) [19]. An example is shown in Figure 2, which contains a binding on line 27. This binding declaration tells Ptolemy to run method `update` when 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 ($\overline{form}$), and a translucid contract (*contract*). These context declarations specify the types and names of reflective information exposed by conforming events [19]. An example is given in Figure 2 on lines 10–18. In writing examples of event types, as in Figure 2, 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 [19]. 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 [7, 19] and we encourage the reader to consult [19].

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 20 of Figure 2 is a method that, when called, will register the method's receiver (**this**). The expression **announce** p `(`$\bar{e}$`)` `{`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 [19]. 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 in an event, or by **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 (`rest` in Figure 2 line 21). 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 4. In this figure, all non-terminals that are used but not defined are the same as in Figure 3.

```
contract ::= requires sp assumes { se } ensures sp
spec     ::= requires sp ensures sp
sp       ::= n | var | sp.f | sp != null | sp == n | sp < n| ! sp
              | sp == old(sp) | sp && sp
se ::= sp | spec | null | new c() | se.m( se ) | se.f | se.f = se| form = se; se
        | if (sp) { se } else { se }| while (sp) { se } | cast c se | se; se
        | register( se ) | invoke( se ) | announce p ( se ) { se }
        |refining spec { se } | next | either { se } or { se }
```

Figure 4: Syntax for writing translucid contracts

A *contract* is of the form **requires** sp_1 **assumes** `{` se `}` **ensures** sp_2. Here, sp_1 and sp_2 are specification predicates as defined in Figure 4 and the body of the contract se is an expression that allows some extra specification-only constructs (such as the choice construct **either** se_T **or** se_F). In an event specification, the predicate sp_1 is the precondition for event announcement, and sp_2 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 2 the method `update` on lines 21–26 must refine the contract of the event type `Changed` on lines 12–17.

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 implementation details (i.e., algorithms) and thus abstracts from a piece of code in a conforming implementation [22, 24]. 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 [22, 24]. In examples we use the following syntactic sugars: **preserves** sp for **requires** sp **ensures** sp, and **establishes** sp for **requires** `1` **ensures** sp [22]. 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** $-$ **or**) are placeholders 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, respectively. The choice expression hides the implementation details 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 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. Choice expression allows variability in handlers' behaviors and enables their abstraction in the translucid contract.

3. VERIFICATION OF PROGRAMS WITH TRANSLUCID CONTRACTS

Verifying Ptolemy 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. Second, if the method is a handler it may call **invoke** that can also cause a set of handlers 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*; an example handler method is `update` in Figure 2.

The main novelty of translucid contracts is that both of 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 types are all the event types that the method is a handler for (as determined by the binding declarations in the class where the handler is declared). 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 body 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 can only 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. 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, imagine we need a sound specification of the behavior of the two handlers `Update` and `Logging` for the event type `Changed` in Figure 2. 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 [8]). 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 $\mathcal{S}$.

Given the specification $\mathcal{S}$ of the behavior of the 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 $\mathcal{S}$ 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`, $\mathcal{S}_{ann}$. 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 code that contains **invoke** expression, once again we start with a specification constructed from event type's contract, e.g., $\mathcal{S}$. 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 $\mathcal{S}$ with the pre- and postcondition of the event type's contract. This gives us a pure and sound OO specification of running two handlers and a correct event body, $\mathcal{S}_{inv}$. 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 the handler refinement.

3.2 Checking Handler Refinement

For sound 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 phase 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 [8], could also be applied.

Figure 5 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 5 checks structural refinement for each handler binding in the program. Rule (CHECK BINDING REF) creates the typing contexts (π, Π) 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 refinement rules in Figure 6 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.

(CLASS TABLE REF)

$$\frac{\forall c \in \mathit{dom}(CT),\ \forall \mathit{binding} \in CT(c) \qquad CT \vdash \mathit{binding}\ \mathbf{in}\ c}{\vdash CT}$$

(CHECK BINDING REF)

$$\frac{\begin{array}{c} \mathit{decl} = t\,\mathbf{event}\,p\,\{t_1\ \mathit{var}_1 \ldots t_n\ \mathit{var}_n\ \mathit{contract}\}, \qquad \mathit{decl} \in CT, \\ \mathit{contract} = \mathbf{requires}\ sp_0\ \mathbf{assumes}\ \{se\}\ \mathbf{ensures}\ sp_1, \\ (t\ m(\mathbf{thunk}\ t'\ \mathit{var}'_0, t'_1\ \mathit{var}'_1 \ldots t'_m\ \mathit{var}'_m)\ \{e\}) \in CT(c), \\ \pi = \{\mathbf{next}\ : \mathbf{thunk}\ t, \mathit{var}_1 : t_1, \ldots, \mathit{var}_n : t_n\}, \\ \Pi = \{\mathbf{this} : c, \mathit{var}'_0 : \mathbf{thunk}\ t', \mathit{var}'_1 : t'_1, \ldots, \mathit{var}'_m : t_m\}, \\ (\pi, \Pi) \vdash se \sqsubseteq e \end{array}}{CT \vdash (\mathbf{when}\ p\ \mathbf{do}\ m)\ \mathbf{in}\ c}$$

Figure 5: Rules for checking structural refinement

A specification expression is refined by a program expression if its subexpressions are refined by corresponding subexpressions of the concrete program expression. Figure 6 shows key rules for checking that. Rules for standard OO expressions are omitted from here, but can be found in our technical report [3]. There is no rule for **register** as it is not allowed in an event type specification. Judgement $(\pi, \Pi) \vdash se \sqsubseteq e$ states that specification expression se is refined by program expression e in the specification typing environment π and program expression typing environment Π, which in turn are constructed in the (CHECK BINDING REF) rule.

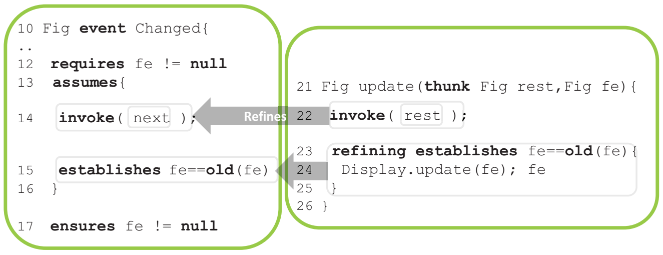
For specification expression se, program expression e, specification and program typing contexts π and Π, se is refined by e, $(\pi, \Pi) \vdash se \sqsubseteq e$, as follows:

Cases of Spec. Exp. (se)	**Refined By (e)**	**Side Conditions**
n	n	
var	var'	**if** $\pi(var) == \Pi(var')$
$sp.f$	$sp'.f$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq sp'$
$sp! = null$	$sp'! = null$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq sp'$
$!sp$	$!sp'$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq sp'$
$sp_1 \&\& sp_2$	$sp'_1 \&\& sp'_2$	**if** $(\pi, \Pi) \vdash sp_1 \sqsubseteq sp'_1$, $(\pi, \Pi) \vdash sp_2 \sqsubseteq sp'_2$
$sp == n$	$sp' == n$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq sp'$
$sp < n$	$sp' < n$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq sp'$
$se_1; se_2$	$e_1; e_2$	**if** $(\pi, \Pi) \vdash se_2 \sqsubseteq e_2$, $(\pi, \Pi) \vdash se_2 \sqsubseteq e_2$
`if`$(sp)\{se_T\}$ **`else`**$\{se_F\}$	**`if`**$(ep)\{e_T\}$ **`else`**$\{e_F\}$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq ep$, $(\pi, \Pi) \vdash se_T \sqsubseteq e_T$, $(\pi, \Pi) \vdash se_F \sqsubseteq e_F$
`while`$(sp)\{se\}$	**`while`**$(ep)\{e\}$	**if** $(\pi, \Pi) \vdash sp \sqsubseteq ep$, $(\pi, \Pi) \vdash se \sqsubseteq e$
$t\ var = se_1; se_2$	$t\ var = e_1; e_2$	**if** $(\pi, \Pi) \vdash se_1 \sqsubseteq e_1$, $\pi' = \pi \uplus \{var : (t, l)\}$, $\Pi' = \Pi \uplus \{var' : (t, l)\}$, $(\pi', \Pi') \vdash se_2 \sqsubseteq e_2$
`refining` $spec\{se\}$	**`refining`** $spec\{e\}$	**if** $(\pi, \Pi) \vdash se \sqsubseteq e$
$spec$	**`refining`** $spec\{e\}$	
`invoke`(se)	**`invoke`**(e)	**if** $(\pi, \Pi) \vdash se \sqsubseteq e$
`announce` $p(\overline{se})$ $\{se\}$	**`announce`** $p(\bar{e})$ $\{e\}$	**if** $(\pi, \Pi) \vdash \overline{se} \sqsubseteq \bar{e}$, $(\pi, \Pi) \vdash se \sqsubseteq e$
`either` $\{se_T\}$ **`or`** $\{se_F\}$	**`if`**$(ep)\{e_T\}$ **`else`**$\{e_F\}$	**if** $(\pi, \Pi) \vdash se_T \sqsubseteq e_T$, $(\pi, \Pi) \vdash se_F \sqsubseteq e_F$
`either` $\{se_T\}$ **`or`** $\{se_F\}$	e_T	**if** $(\pi, \Pi) \vdash se_T \sqsubseteq e_T$
`either` $\{se_T\}$ **`or`** $\{se_F\}$	e_F	**if** $(\pi, \Pi) \vdash se_F \sqsubseteq e_F$

Figure 6: Structural refinement relation ($\sqsubseteq$)

3.2.1 *Example Handler Refinement*

To illustrate the refinement rules in Figure 6, consider checking whether the handler method `update` on lines 22–25 in Figure 2 refines the translucid contract's body on lines 14–15. As illustrated in Figure 7 and according to the rule for $se_1; se_2$ in Figure 6, 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}`.



```
10 Fig event Changed{
..
12   requires fe != null
13   assumes{
14    invoke( next );
15    establishes fe==old(fe)
16   }
17   ensures fe != null
```

```
21 Fig update(thunk Fig rest,Fig fe){
22 invoke( rest );
23  refining establishes fe==old(fe){
24   Display.update(fe); fe
25  }
26 }
```

Figure 7: Handler refinement

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 Π). 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 Π. By applying the rule for $spec$ in Figure 6, 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 the event type `Changed`.

The refinement rule for the 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.

3.3 Verifying Ptolemy Programs

The main difficulty in verifying Ptolemy programs is that **`announce`** and **`invoke`** expressions could cause a statically unknown set of handlers (advice) to run. This set is not known statically unless a whole program analysis is performed. Thus such knowledge is not part of modular verification. Despite this, translucid contracts make modular verification possible. The challenge is to verify the code containing **`announce`** and **`invoke`** expressions. The basic idea is to use the translucid contract of the event type in place of each handler as discussed in Section 3.1. There are two types of methods to verify, regular methods which might announce event and handler method which handle the events.

3.3.1 *Verification of Regular Methods*

To statically verify a non-handler method $t\ m\ (\bar{t}\ \overline{var})\{e\}$ 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 8 shows how to do that. Basically, a translation function $Tr(se, b_e, p)$ 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, modeling event announcement. Event p is the announced event, if any, and b_e 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 $Tr(e, \textbf{skip}, \perp)$. The arguments **`skip`** and $\perp$ specify that this method does not handle any events ($\perp$) and thus there is no event body (**`skip`**) which basically means the method is a non-handler. These parameters are included in this case simply to facilitate uniform application of the Tr function for both regular (non-handler) and handler methods. **`skip`** is sugar for **`while`** `0 { 0 }`.

The result of $Tr(e, \textbf{skip}, \perp)$ is a specification expression with no Ptolemy-specific features, but may have extra expressions which simulate event announcement and running of handlers. This expression can then be used to perform standard weakest precondition based verification for OO programs.

3.3.2 *Verification of Handler Methods*

To statically verify a handler method h of the form $t\ h\ (\textbf{thunk}\ t_0\ var_0, \bar{t}\ \overline{var})\ \{e\}$, for each event type p with a binding **`when`** p **`do`** h, one does the following. Let the contract for p be **`requires`** sp_p **`assumes`** $\{se_p\}$ **`ensures`** sp'_p, then compute $Tr(e, \textbf{requires}\ sp_p\ \textbf{ensures}\ sp'_p, p)$ and use the result to verify the handler h. The second argument to Tr is a specification statement consisting of the event's pre- and postconditions; this is used in the place of the announced event's body, since the event body is not available during static verification of the handler, and since this specification statement must be refined by all event bodies. The result of $Tr(e, \textbf{requires}\ sp_p\ \textbf{ensures}\ sp'_p, p)$ is a pure OO specification expression.

For specification expressions se, expressions b_e, event types p,
where p has contract $\texttt{\textbf{requires}}\ sp_p\ \texttt{\textbf{assumes}}\ \{se_p\}\ \texttt{\textbf{ensures}}\ sp'_p$
and context variables $\overline{t}\ \overline{var}$,
$Tr(se, b_e, p) =$

Cases of se	Result	Side Condtions
n, $\texttt{\textbf{var}}$, $\texttt{\textbf{null}}$, $\texttt{\textbf{new}}\ c()$, $\texttt{\textbf{next}}$, $spec$	se	
$\texttt{\textbf{old}}\ (se_1)$	$\texttt{\textbf{old}}\ (se_2)$	**if** $se_2 = Tr(se_1, b_e, p)$
$se_1.f$	$se_2.f$	**if** $se_2 = Tr(se_1, b_e, p)$
$\texttt{\textbf{either}}\ \{se_0\}\ \texttt{\textbf{or}}\ \{se_1\}$	$\texttt{\textbf{either}}\ \{se'_0\}\ \texttt{\textbf{or}}\ \{se'_1\}$	**if** $se'_0 = Tr(se_0, b_e, p)$, $se'_1 = Tr(se_1, b_e, p)$
$se.m(\overline{se})$	$se'.m(\overline{se}')$	**if** $se' = Tr(se, b_e, p)$, $\overline{se'} = Tr(\overline{se}, b_e, p)$
$se_0.f = se_1$	$se'_0.f = se'_1$	**if** $se'_0 = Tr(se_0, b_e, p)$, $se'_1 = Tr(se_1, b_e, p)$
$\texttt{\textbf{if}}(ep)\{se_0\}\ \texttt{\textbf{else}}\{se_1\}$	$\texttt{\textbf{if}}(ep')\{se'_0\}\texttt{\textbf{else}}\{se'_1\}$	**if** $ep' = Tr(ep, b_e, p)$, $se'_0 = Tr(se_0, b_e, p)$, $se'_1 = Tr(se_1, b_e, p)$
$\texttt{\textbf{while}}(ep)\{se\}$	$\texttt{\textbf{while}}(ep')\{se'_0\}$	**if** $ep' = Tr(ep, b_e, p)$, $se' = Tr(se, b_e, p)$
$\texttt{\textbf{cast}}\ c\ se$	$\texttt{\textbf{cast}}\ c\ se'$	**if** $se' = Tr(se, b_e, p)$
$se_0; se_1$	$se'_0; se'_1$	**if** $se'_0 = Tr(se_0, b_e, p)$, $se'_1 = Tr(se_1, b_e, p)$
$t\ var = se_0; se_1$	$t\ var = se'_0; se'_1$	**if** $se'_0 = Tr(se_0, b_e, p)$, $se'_1 = Tr(se_1, b_e, p)$
$\texttt{\textbf{refining}}\ spec\ \{se_1\}$	$spec$	
$\texttt{\textbf{register}}(se_1)$	se_2	**if** $se_2 = Tr(se_1, b_e, p)$
$\texttt{\textbf{invoke}}(se_1)$	$\texttt{\textbf{refining}}\ spec\{$ $\texttt{\textbf{either}}\ \{se_2; b_e\}$ $\texttt{\textbf{or}}\ \{se_2; se_3\}$ $\}$	**if** $se_2 = Tr(se_1, b_e, p)$, $se_3 = Tr(se_p, b_e, p)$, $spec = \texttt{\textbf{requires}}\ sp_p\ \texttt{\textbf{ensures}}\ sp'_p$
$\texttt{\textbf{announce}}\ p'\ (\overline{se})\ \{se_1\}$	$\texttt{\textbf{refining}}\ spec\{$ $\texttt{\textbf{either}}\ \{\overline{se'}; se'_1\}$ $\texttt{\textbf{or}}\ \{\overline{t'}\ \overline{var'} = \overline{se'}; se'_2\}$ $\}$	**if** p' has translucid contract $\texttt{\textbf{requires}}\ sp_{p'}\ \texttt{\textbf{assumes}}\ \{se_{p'}\}\ \texttt{\textbf{ensures}}\ sp'_{p'}$ and context variables $\overline{t'}\ \overline{var'}$, $\overline{se'} = Tr(\overline{se}, b_e, p)$, $se'_1 = Tr(se_1, b_e, p')$, $se'_2 = Tr(se_{p'}, se'_1, p')$, $spec = \texttt{\textbf{requires}}\ sp_{p'}\ \texttt{\textbf{ensures}}\ sp'_{p'}$

Figure 8: Translation algorithm. This is the algorithm for converting program expressions into specification expressions that simulate running of handlers.

3.3.3 Translation Function

As illustrated in Section 3.1, the translation function $Tr(se, b_e, p)$, with p as the announced event and b_e as the body of p, inlines event type specification or event body in the place of announce and invoke expressions in se, and computes a simulating specification of the event announcement. Announce and invoke expressions are replaced by the event type's contract if there are more applicable handlers and are replaced by the event body otherwise. As existence or non-existence of more applicable handlers is not decidable statically, the translation algorithm considers occurrence of both of these situations simultaneously using an $\texttt{\textbf{either}} - \texttt{\textbf{or}}$ choice expression, as shown in Figure 8.

Most cases in the translation function Tr are straightforward as they just recursively apply Tr to their subexpressions and compose the results. Translations of refining, announce and invoke expressions are of more interest, though. Translation of $\texttt{\textbf{refining}}\ spec\ \{e\}$ is $spec$ as the runtime assertion checking ensures that e refines the $spec$. The cases for invoke and announce expressions are central as they model event announcement by simulating running of the handlers and the event body.

Translations of invoke and announce expressions, both produce an $\texttt{\textbf{either}} - \texttt{\textbf{or}}$ choice expression guarded by a $\texttt{\textbf{refining}}$ expression. The either-branch simulates the situation when there is no applicable handler whereas the or-branch takes care of the situation when there exist more handlers to run.

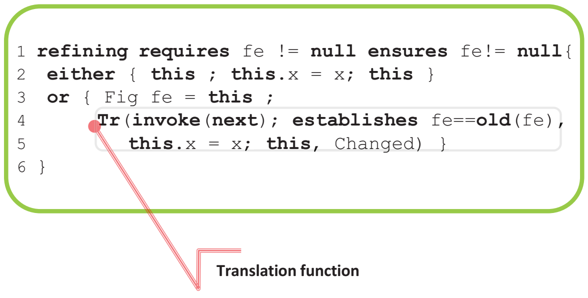
In the translation of $\texttt{\textbf{invoke}}\ (se_1)$, the either-branch contains a sequence of two expressions: translation of the argument se_1 and the event body b_e, which means no more handler to run. The or-branch contains a sequence of two expressions too: translation of argument se_1 and translation of the translucid contract se_p. The guarding refining expression assures that specification $spec$ is satisfied by the choice expression inside. $spec$ contains pre- and postcondition of the contract se_p.

Translation of announce expression is similar to the invoke. In case of $\texttt{\textbf{announce}}\ p'\ (\overline{se})\ \{se_1\}$, the either-branch contains a sequence of two expressions: translation of the argument $\overline{se}$ and the translation of event body se_1. In or-branch the first expression is the translation of the arguments and their assignment them to context variables $\overline{var'}$. The second expression is the translation of the translucid contract of event p', i.e. $se_{p'}$, assuming that the event body is se'_1, the translation of se_1. The translation of $se_{p'}$ simulates running of handlers for event p' with a concrete event body and event type's translucid contract as an abstraction for handlers.

The translation function assumes an acyclic event announce/handle relation. Circular relations could simply be detected statically.

3.3.4 Illustration of the Verification Algorithms

To illustrate, consider verifying the method `setX` in Figure 1 with the translucid contract in Figure 2. The body of this method is the announce expression **`announce`** `Changed(`**`this`**`){` **`this`**`.x = x;` **`this`**`}`. To verify this method, we first apply the translation function $Tr(se, \texttt{\textbf{skip}}, \perp)$ with $se = \texttt{\textbf{announce}}$ $\texttt{Changed}(\texttt{\textbf{this}})\{\texttt{\textbf{this}}.x = x; \texttt{\textbf{this}}\}$ as this method is a non-handler regular method. The case for announce expression in Figure 8 is applicable, which results in the specification expression shown in Figure 9.

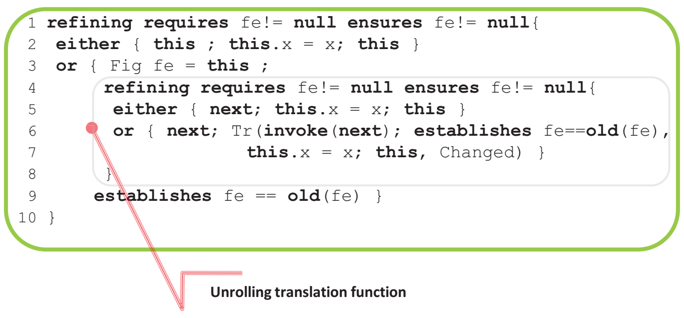


```
1 refining requires fe != null ensures fe!= null{
2  either { this ; this.x = x; this }
3  or { Fig fe = this ;
4       Tr(invoke(next); establishes fe==old(fe),
5          this.x = x; this, Changed) }
6 }
```

Figure 9: Translation of method setX

Notice the use of the translation function Tr on lines 4–5. To verify this expression both the either-branch and the or-branch must be verified. During the verification, upon reaching the translation function, it is unrolled one more time resulting in the specification expression shown in Figure 10.

During this application, the cases for sequence, $spec$ and invoke expressions are used, which again results in an embedded translation function Tr on lines 6–7. The astute readers may have observed that we have essentially reduced problem of verifying $\texttt{\textbf{announce}}$ and $\texttt{\textbf{invoke}}$ expressions to a problem similar to reasoning about loops. Thus, standard techniques for reasoning about



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

Figure 10: Unrolling translation function

loops, such as proof rules that rely on user-supplied invariants, could be applied here. Heuristics like the one used in ESC/Java [8] to unroll the loops are also applicable here. When the verifier decides to terminate recursive unrolling, based on any of the above-mentioned approaches, the translation function in the result expression is just ignored. Verification of the method `update` is similar.

3.3.5 *Soundness*

To prove the soundness of our verification and reasoning approach, we have proved the translation algorithm sound, i.e., the specification expression (produced by the translation algorithm which simulates event announcement) and used for verification is refined by the program expression obtained after recursive replacement of event announcements by concrete handler method bodies. More details are presented in our companion technical report [3].

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) each handler method satisfies the specification of the event types it handles (2) each event body satisfies the pre- and postconditions of its event type specification, (3) each **`refining`** expression body refines the specification it claims to refine, and (4) each event announcement and consequent execution of all of its handler methods combined behavior, satisfies pre- and postconditions of the event type, regardless of the number of the handlers and their order of execution. Alternatively, a static checker like ESC/Java [8] could discharge these assumptions.

We have implemented runtime assertion checking in the Ptolemy compiler [18]. Figure 11 illustrates insertion of runtime probes by the Ptolemy compiler in the generated code. An abstraction function matches up context variable `fe` to its corresponding variables in the scopes of subject `Point` and handler `Update`.

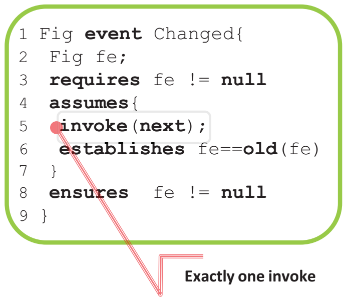
To meet obligation (1) pre- and postcondition probes are inserted at the beginning and end of handler method body, before line 21 and after line 26. Runtime probes right before and after line 6 guarantee obligation (2). To verify that the refining expression on lines 23-25 refines the specification it claims to refine, obligation (3), runtime assertions are inserted before line 23 and after line 25. 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 **`announce`** and **`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 22 meet the obligations for invoke expressions.

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 *et al.* [23]. Rinard *et al.* classify base-advice 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 **`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 advice which call **`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 the direct interference.

4.1 Direct Interference: Augmentation

Informally an augmentation handler evaluates **`invoke`** expression exactly once. An augmentation handler can be a before or after handler. After-augmentation handler is executed after the event body whereas in the before augmentation the order is opposite.



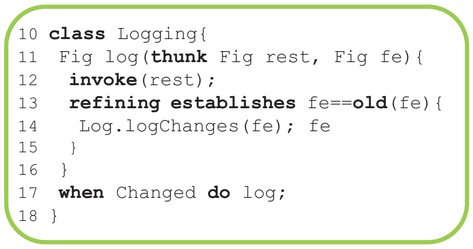
```
1 Fig event Changed{
2  Fig fe;
3  requires fe != null
4  assumes{
5   invoke(next);
6   establishes fe==old(fe)
7  }
8  ensures  fe != null
9 }
```

Figure 12: Specifying augmentation with a translucid contract

To illustrate consider the translucid contract in Figure 12 on lines 3–8. Translucid contracts are required to reveal all appearances of the invoke expression, thus it is assured that all refining handlers will evaluate invoke expression exactly once.

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

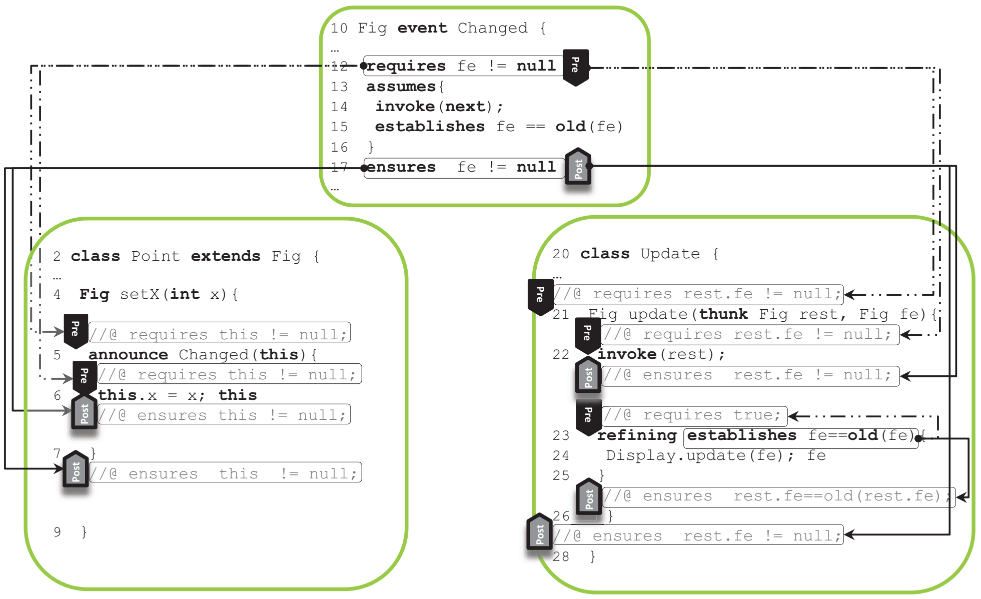
Method `log` in class `Logging` in Figure 13 is an example of a conforming after-augmentation handler. The requirement for this method is "to log the changes when figures are changed". The handler `log` causes the event body to be run first by calling **`invoke`** on line 12 and then logs the changes in the figure on line 14. The classes `Point` and `Fig` are the same as in Figure 1.



```
10 class Logging{
11  Fig log(thunk Fig rest, Fig fe){
12   invoke(rest);
13   refining establishes fe==old(fe){
14    Log.logChanges(fe); fe
15   }
16  }
17  when Changed do log;
18 }
```

Figure 13: After-augmentation handler

Structural similarity requires the handler implementation to evaluate **`invoke`** exactly once and at its very beginning which in turn



```
10 Fig event Changed {
…
12  requires fe != null
13  assumes{
14   invoke(next);
15   establishes fe == old(fe)
16  }
17  ensures  fe != null
…
```

```
2 class Point extends Fig {
…
4  Fig setX(int x){
    //@ requires this != null;
5   announce Changed(this){
     //@ requires this != null;
6    this.x = x; this
     //@ ensures this != null;
7   }
    //@ ensures  this  != null;

9  }
```

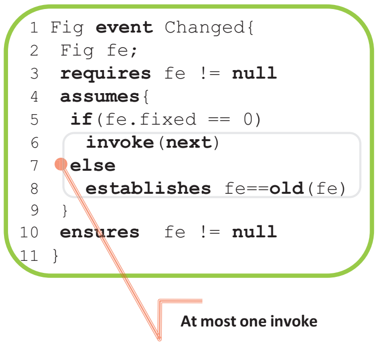
```
20 class Update {
…
   //@ requires rest.fe != null;
21  Fig update(thunk Fig rest, Fig fe){
     //@ requires rest.fe != null;
22   invoke(rest);
     //@ ensures  rest.fe != null;

     //@ requires true;
23   refining establishes fe==old(fe){
24    Display.update(fe); fe
25   }
     //@ ensures  rest.fe==old(rest.fe);
26  }
   //@ ensures  rest.fe != null;
28  }
```

Figure 11: Runtime assertion checking (RAC). Gray lines show pseudo code corresponding to generated code by the compiler.

ensures that the handlers is an "after-augmentation" handler. The handler refines the contract because line 12 matches line 5 and the refining expression on lines 13–15 refines the same specification as on line 6.

4.2 Direct Interference: Narrowing

A narrowing handler evaluates **invoke** at most once, which implies existence of a conditional statement guarding **invoke**.



```
1 Fig event Changed{
2  Fig fe;
3  requires fe != null
4  assumes{
5   if(fe.fixed == 0)
6     invoke(next)
7   else
8     establishes fe==old(fe)
9  }
10 ensures  fe != null
11 }
```

Figure 14: Specifying narrowing with a translucid contract

To illustrate consider the translucid contract in Figure 14 on lines 5–8 which specifies narrowing handlers. The contract reveals appearances of **invoke** expression and the **if** expression guarding that which in turn ensures that invoke expression is evaluated at most once. It does not, however, reveal the actual code that must refine the specification on line 8. All the refining handlers will have the same structure in their implementation with regard to invoke and if expressions, which makes them narrowing handlers.

Figure 15 illustrates a narrowing handler refining the contract shown in Figure 14. 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 `fixed` is added to the class `Fig`, line 23. For fixed figures the value of this field is 1 and 0 otherwise. The class `Point` is the same as in Figure 1. To implement the constraint the handler `check` skips invoking the base code whenever the figure is fixed (checked by accessing the field `fixed`).

```
12 class Enforce{
13  Fig check(thunk Fig rest, Fig fe){
14   if(fe.fixed == 0)
15    invoke(rest)
16   else
17    refining establishes fe==old(fe){
18     fe
19    }
20  }
21  when Changed do check;
22 }
```

```
23 class Fig { int fixed; }
```

Figure 15: Narrowing handler

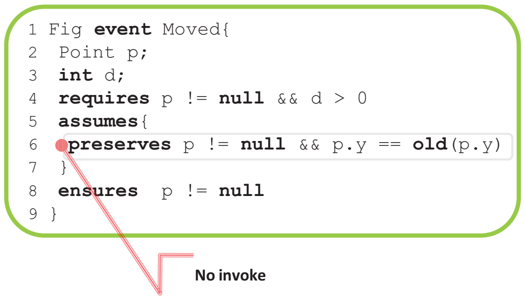
For the handler `check` to refine the contract in the event type `Changed`, its implementation must structurally match the contract. The true block of the **if** expression on line 14–15 refines the true block of the **if** on lines 5–6 as they textually match. The false block of the **if** on line 16–19 refines the false block of the **if** on lines 7–8 because lines 17–19 claim to refine the specification on line 8. This claim is discharged by runtime assertions.

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.

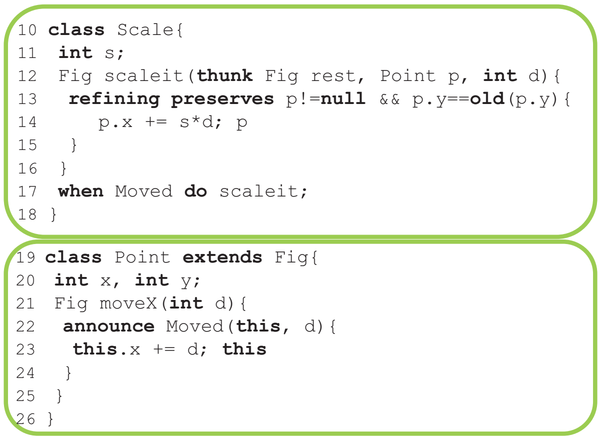
Figure 16 shows the contract in event type `Moved` specifying replacement handlers by not evaluating any **invoke** expression in the contract, line 6. Notice that (non) existence of an invoke expression in the contract *requires* the handler implementation to (not) evaluate the invoke in its body.

Figure 17 shows a replacement handler refining the contract in Figure 16. The example uses several standard sugars such as `+=`



```
1 Fig event Moved{
2  Point p;
3  int d;
4  requires p != null && d > 0
5  assumes{
6   preserves p != null && p.y == old(p.y)
7  }
8  ensures  p != null
9 }
```

Figure 16: Specifying replacement with a translucid contract



```
10 class Scale{
11  int s;
12  Fig scaleit(thunk Fig rest, Point p, int d){
13   refining preserves p!=null && p.y==old(p.y){
14      p.x += s*d; p
15   }
16  }
17  when Moved do scaleit;
18 }
19 class Point extends Fig{
20  int x, int y;
21  Fig moveX(int d){
22   announce Moved(this, d){
23    this.x += d; this
24   }
25  }
26 }
```

Figure 17: Replacement handler

and >. 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 s, defined in class Scale".

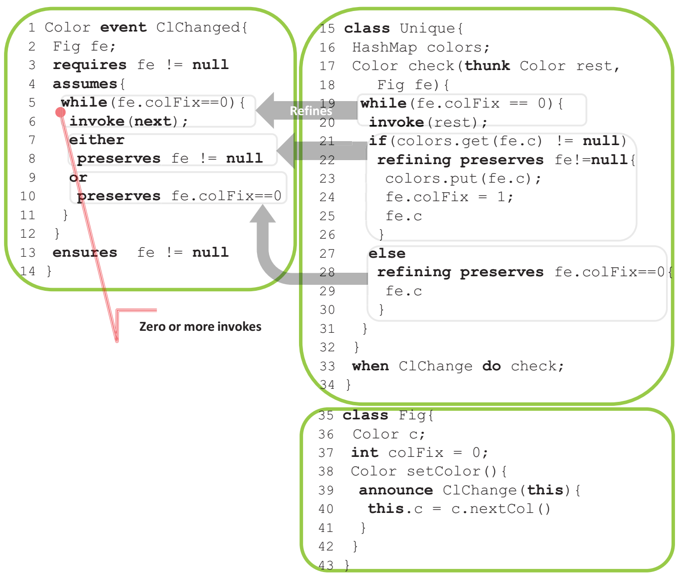
If an contract has no **invoke** expression, none of the refining handlers are allowed to have an **invoke** in their implementation. Otherwise the structural similarity criterion of the refinement is violated. The handler scaleit refines Moved's contract because its body on lines 13–15 matches the specification on line 6.

4.4 Direct Interference: Combination

A Combination handler, typically useful for fault tolerance, can functionalities, can evaluate **invoke** expression any number of times. Figure 18 illustrates a combination contract and a handler. 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 guarding the **invoke** expression by **while**. Based on the contract specially looking at the while loop surrounding invoke, the base code developer can conclude that handler methods for event ClChange may run the original event body multiple times. The developer, however, is not aware of the concrete details of handlers, thus those details remain hidden.

A combination handler is illustrated in Figure 18 lines 15–34. In this example, colors are added to the figures elements by adding a field color to the class Fig and by providing a method setColor for picking the color of the figure, lines 35–43. The class Color which provides a method nextCol to get the next available color is not shown.

To implement the requirement that "each figure should have a unique color", event type ClChange is declared as an abstraction



```
1 Color event ClChanged{
2  Fig fe;
3  requires fe != null
4  assumes{
5   while(fe.colFix==0){
6    invoke(next);
7    either
8     preserves fe != null
9    or
10    preserves fe.colFix==0
11   }
12  }
13  ensures  fe != null
14 }
```

```
15 class Unique{
16  HashMap colors;
17  Color check(thunk Color rest,
18     Fig fe){
19  while(fe.colFix == 0){
20   invoke(rest);
21   if(colors.get(fe.c) != null)
22    refining preserves fe!=null{
23     colors.put(fe.c);
24     fe.colFix = 1;
25     fe.c
26    }
27   else
28    refining preserves fe.colFix==0{
29     fe.c
30    }
31   }
32  }
33  when ClChange do check;
34 }
35 class Fig{
36  Color c;
37  int colFix = 0;
38  Color setColor(){
39   announce ClChange(this){
40    this.c = c.nextCol()
41   }
42  }
43 }
```

Figure 18: Combination contract and handler

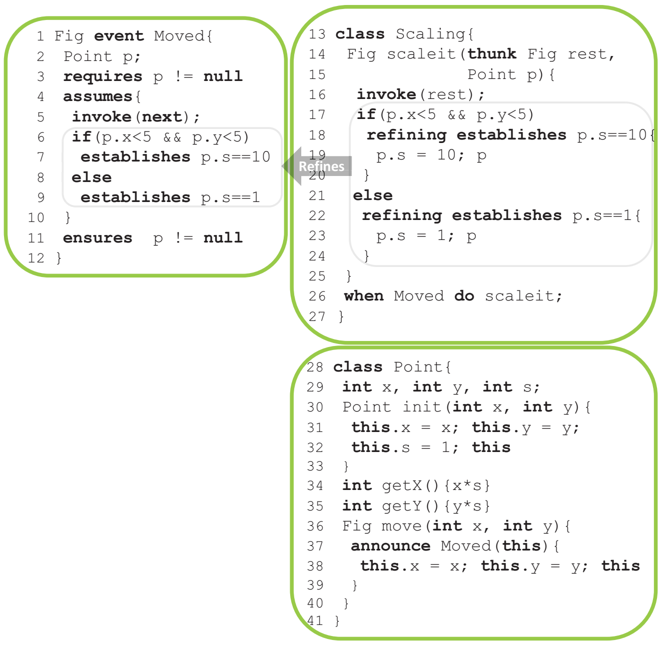
of events representing colors changes. The method setColor changes colors so it announces the event ClChange on lines 39–41. The body of the announce expression contains the code to obtain the next color on line 40. The handler Unique on lines 15–34 implements this requirement by storing already-used 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 then it 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 into the hash table and colFix is set to 1, lines 21–26.

The specification for ClChange on lines 4–12 says that a combination handler will be run when this event is announced. The specification makes use of the choice feature, on line 7–10. To correctly refine the specification, based on the refinement rules in Figure 6, a handler can either have a refining **if** expression at the corresponding place in its body or it can have an unconditional expression refining the either-block or the or-branch in the specification. Refinement between specification and implementation blocks is illustrated in the figure.

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 [23]. However, it can not be used to analyze the possibility of two or more control flow paths each of which being, e.g. an augmentation, if each path maintains a different invariant. Figure 19 illustrates such a scenario with an example adapted from [13].

In this example the requirement is "a point should be visibly distinguished from the origin" [13]. If the point is close enough to the origin, its coordinates will be scaled up by a scaling factor s added to Point on line 29, initially set to 1, line 32. The scaling factor s has only two values: 1 and 10. The requirement is implemented in the handler method scaleit which runs whenever event Moved



```
1  Fig event Moved{
2   Point p;
3   requires p != null
4   assumes{
5    invoke(next);
6    if(p.x<5 && p.y<5)
7     establishes p.s==10
8    else
9     establishes p.s==1
10  }
11  ensures  p != null
12 }
```

```
13 class Scaling{
14  Fig scaleit(thunk Fig rest,
15                   Point p){
16   invoke(rest);
17   if(p.x<5 && p.y<5)
18    refining establishes p.s==10{
19     p.s = 10; p
20    }
21   else
22    refining establishes p.s==1{
23     p.s = 1; p
24    }
25  }
26  when Moved do scaleit;
27 }
```

```
28 class Point{
29  int x, int y, int s;
30  Point init(int x, int y){
31   this.x = x; this.y = y;
32   this.s = 1; this
33  }
34  int getX(){x*s}
35  int getY(){y*s}
36  Fig move(int x, int y){
37   announce Moved(this){
38    this.x = x; this.y = y; this
39   }
40  }
41 }
```

Figure 19: Expressive control flow properties beyond [23]

is announced and sets up the scaling factor to 10 if the point is close enough to the origin (vicinity condition). The vicinity condition is true if the point's x and y coordinates are both less than 5. The class Fig is the same as in Figure 1.

The assertions to be validated here are as follows: (i) all of the handlers are after-augmentation ones, (ii) the scaling factor s is either 1 or 10, and (iii) sis 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). However translucid contracts 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 three of the above-mentioned assertions.

In summary, in this section we show that translucid contracts enable specification of control flow interference between a subject and its observers and allow automatic enforcement of specified interference patterns via structural refinement. Translucid contracts are expressive enough to specify and enforce control interference properties proposed by Rinard *et al.* and even the more sophisticated ones which could not be specified by previous works on the design by contract for aspects.

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 specification.

5.1 Contracts for Aspects

This work is closest in the spirit to the work on crosscutting programming interfaces (XPIs) [27]. 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 behavioral contracts and thus are not easily checked automatically. Furthermore, these works do not describe a verification technique and contracts could be bypassed.

Skotiniotis and Lorenz [25] 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 [30]. 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 black box contracts and thus may not be used to reason about control effects of advice.

5.2 Modular Reasoning

There is a large body of work on modular reasoning about AO programs on language designs [1, 7, 10], design methods [14, 27], and verification techniques [11, 15]. 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 [4,9], our approach based on structural refinement is significantly lightweight. Furthermore, it accounts for quantification that these ideas do not.

Oliveira *et al.* [17] 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 [10] and Steimann *et al.*'s join point types [26] share similar spirit with Rajan and Leavan's event types [19]. Although Steimann *et al.* proposed informal behavioral specification, their work has no explicit notion of formally expressed and enforced contracts, or stating interaction behavior, nor do any of these other approaches.

The work of Khatchadourian *et al.* [12] 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.

5.3 Grey Box Specification and Verification

This work builds upon previous research on grey box specification and verification [6]. Among others, Barnett and Schulte have used grey box specifications written in AsmL [5] for verifying contracts for .NET, Wasserman and Blum [29] also use a restricted form of grey box specifications for verification, Tyler and Soundarajan [28] and most recently Shaner *et al.* [24] 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 specification to enable expressive assertions about web-services [22]. 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.

6. 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, Ptolemy [19] 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, 10, 14, 26, 27] in our technical report [3].

Second, Ptolemy's explicit announcement solves the problem of frequent join point shadows, since one only has to deal with handlers where events are explicitly announced.

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 [18].

Adding translucid contracts to other AO compilers, integrating our ideas with the rich specification features of JML, and working out larger examples to find out more of the practical use cases of translucid contracts are some directions for future work. Another direction is to use translucid contracts to reason about *data effects* of subject-observer interaction patterns.

Acknowledgments

Rajan and Bagherzadeh were supported in part by the NSF grants CCF-08-46059 and CCF-10-17334. Leavens and Rajan were supported in part by NSF grants CNS-08-08913.

7. REFERENCES

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

[2] M. Bagherzadeh, H. Rajan, and G. T. Leavens. Translucid contracts for aspect-oriented interfaces. In *FOAL '10*.

[3] M. Bagherzadeh, H. Rajan, G. T. Leavens, and S. Mooney. Translucid contracts for aspect-oriented interfaces. Technical Report 10-02, Iowa State U., Dept. of Computer Sc., 2009.

[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. Runtime verification of .NET contracts. *Journal of Systems and Software*, 65(3), 2003.

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

[7] C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *SCP '06*, 63(3).

[8] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02*.

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

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

[11] S. Katz. Diagnosis of harmful aspects using regression verification. In *FOAL '04*.

[12] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving aspect-oriented programs. In *FOAL '08*.

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

[14] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05*, pages 49–58.

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

[16] J. M. Morris. A theoretical basis for stepwise refinement and the programming calculus. *Sci. Com. Program.*, 9(3), 1987.

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

[18] Ptolemy with Translucid Contracts. `http://www.cs.iastate.edu/~ptolemy/contract/`.

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

[20] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*.

[21] H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *TOSEM '08*.

[22] 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*.

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

[24] 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*.

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

[26] F. Steimann, T. Pawlitzki, S. Apel, and C. Kastner. Types and modularity for implicit invocation with implicit announcement. *TOSEM '10*, 20(1).

[27] K. J. Sullivan, W. G. Griswold, H. Rajan, Y. Song, Y. Cai, M. Shonle, and N. Tewari. Modular aspect-oriented design with XPIs. *TOSEM '09*, 20(2).

[28] B. Tyler and N. Soundarajan. Black-box testing of grey-box behavior. In *FATES '03, 1–14.*

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

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