2011

Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Megdi Bagherzadeh
Iowa State University

Gary T. Leavens
University of Central Florida

Sean Moody
Iowa State University

Follow this and additional works at: https://lib.dr.iastate.edu/cs_conf

Part of the Computer Sciences Commons

Recommended Citation
https://lib.dr.iastate.edu/cs_conf/21

This Conference Proceeding is brought to you for free and open access by the Computer Science at Iowa State University Digital Repository. It has been accepted for inclusion in Computer Science Conference Presentations, Posters and Proceedings by an authorized administrator of Iowa State University Digital Repository. For more information, please contact digirep@iastate.edu.
Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces

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.

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

Disciplines
Computer Sciences

Comments

Rights
© ACM, 2011 This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in Proceedings of the tenth international conference on Aspect-oriented software development, pp. 141-152. ACM, 2011. https://doi.org/10.1145/1960275.1960293

This conference proceeding is available at Iowa State University Digital Repository: https://lib.dr.iastate.edu/cs_conf/21
Translucid Contracts: Expressive Specification and Modular Verification for Aspect-Oriented Interfaces

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

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 \( x \).

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

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   ...
28 }
27  when 
Quantification 
Update init(){
   thunk
); fe 
update;
Registration 
Black Box 
Contract
30 class Logging{
31  ...
32  Fig log( thunk Fig rest, Fig fe){
33   invoke (rest); 
34   Log.logChanges(fe); fe 
35  }
36  when Changed do log ;
37 }
38
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
occurrence of the \texttt{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 \texttt{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 \texttt{setX} on line 6 of Figure 1 will always be run. Such conclusions allow a client of the \texttt{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 \texttt{next} is a specification placeholder for the event closure passed to the handlers.

Requiring the \texttt{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 \texttt{update} on lines 22–25 in Figure 2 redefines 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 \texttt{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 how 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-innovation (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{verbatim}
prog ::= decl e
decl ::= class : extends d | field m binding
field ::= | event p (form contract)
form ::= t var | when var | this and var | next
binding ::= when y do m
e ::= n | var | null | new c() | e, m(T) | e.f | e.f = e | form = e | e
| if (ep) | else (e) | while (ep) | c e | cast e | e
| register (e) | invoke (e) | announce p (T) | e
| refining spec (e)
ep ::= n | var | ep.f | ep != null | ep == n | ep < n | ! ep | ep & ep
where
n ∈ N, the set of numeric, integer literals
c, d ∈ C, a set of class names
t ∈ C ∪ \{int\}, a set of types
p ∈ P, a set of event type names
f ∈ F, a set of field names
m ∈ M, a set of method names
var ∈ \{this, next\} ∪ V, V is a set of variable names
\end{verbatim}

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

2.1.1 Declarations

We do not allow nesting of \texttt{decls}. A class has a name (c) and names its superclass (d), and may declare fields (\texttt{field}) and methods (\texttt{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 (form), and
a translucent contract (contract). These context declarations specify
the types and names of reflective information exposed by conform-
ing 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 for-
amal 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 lan-
guage [19]. It includes several standard object-oriented (OO) ex-
pressions and also some expressions that are specific to announc-
ing 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 [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 (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 cor-
responding 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 imple-
ment 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.

A contract is of the form requires sp1 assumes { se } ensures sp2. Here, sp1 and sp2 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 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 specifi-
cation 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 vari-
ables. 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 con-
tracts: specification expressions, next expressions, abstract
invoke expressions, and choice expressions. A specification ex-
pression (spec) hides implementation details (i.e., algorithms) and
thus abstracts from a piece of code in a conforming implementa-
tion [22, 24]. 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 implement-
ment contains a refining expression whose body expression,
when started in a state that satisfies sp1, will terminate in a
state that satisfies sp2 [22, 24]. In examples we use the following
typographic 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 expres-
sion (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 con-
crete condition check in the handler method. For a choice expres-
sion either { se1 } or { se2 } a conforming handler may con-
tain an expression e1 that refines se1, or an expression e2 that re-
fines se2, or an expression if { e0 } { e1 } else { e2 }, where
e0 is a side-effect free expression, e1 refines se1, and e2 refines
se2. Choice expression allows variability in handlers’ behaviors
and enables their abstraction in the translucid contract.

3. VERIFICATION OF PROGRAMS WITH
TRANSCLUDIC CONTRACTS

Verifying Ptolemy programs is different from standard object-
oriented (OO) programs in two ways. First, a method in the pro-
gram 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 con-
tracts poses two novel technical problems, compared to verifying
standard OO programs: (1) verifying that each handler method cor-
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 de-
claration, 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 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 $S$.

Given the specification $S$ 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 $S$ 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, $S_{\text{inv}}$. 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., $S$. 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 $S$ with the pre- and post-condition of the event type’s contract. This gives us a pure and sound OO 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 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 $(\pi, \Pi)$ 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 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.

\begin{align*}
\text{(CLASS TABLE REF)} & \quad \forall c \in \text{dom}(CT), \forall \text{binding} \in CT(c) \quad CT \vdash \text{binding in } c \\
\text{(CHECK BINDING REF)} & \quad \text{decl = } \{ \text{event } p \{ t_1 : \text{var}_1, \ldots, t_n : \text{var}_n : \text{contract} \}, \text{ decl } \in CT, \\
& \quad \text{contract = requires } \text{sp}_0 \text{ ensures } \text{sp}_1, \quad (t_m : \text{thunk } t' : \text{var}_1, \ldots, \text{var}_m : t_m) \in \text{CT}(c), \\
& \quad \pi = \{ \text{next : thunk } t' : \text{var}_1, \ldots, \text{var}_m : t_m \}, \\
& \quad \Pi = \{ \text{this } : c, \text{var}_0 : \text{thunk } t', \text{var}_1 : t_1, \ldots, \text{var}_m : t_m \}, \\
& \quad (\pi, \Pi) \vdash se \subseteq c \\
\text{CT} \vdash \text{(when } p \text{ do } m) \text{ in } c
\end{align*}

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 $(\pi, \Pi) \vdash se \subseteq c$ 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.
For specification expression \( se \), program expression \( e \), specification and program typing contexts \( \pi \) and \( \Pi \), \( se \) is refined by \( e \), \( (\pi, \Pi) \mapsto se \subseteq e \), as follows:

<table>
<thead>
<tr>
<th>Cases of Spec. Exp. (se)</th>
<th>Refined By (e)</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n \in \pi )</td>
<td>( n \in \Pi )</td>
<td></td>
</tr>
<tr>
<td>( \text{var} )</td>
<td>( \text{var}' )</td>
<td>( \pi \mapsto \Pi \mapsto \text{var} \mapsto \text{var}' )</td>
</tr>
<tr>
<td>( \text{sp}, \text{f} )</td>
<td>( \text{sp}', \text{f} )</td>
<td>( \pi \mapsto \Pi \mapsto \text{sp} \mapsto \text{sp}' )</td>
</tr>
<tr>
<td>( \text{sp} = \text{null} )</td>
<td>( \text{sp}' = \text{null} )</td>
<td>( \pi \mapsto \Pi \mapsto \text{sp} \mapsto \text{sp}' )</td>
</tr>
<tr>
<td>( t ) ( \text{sp}, \text{k} &amp; \text{sp}_2 )</td>
<td>( t ) ( \text{sp}', \text{k} &amp; \text{sp}_2 )</td>
<td>( \pi \mapsto \Pi \mapsto \text{sp} \mapsto \text{sp}_1 )</td>
</tr>
<tr>
<td>( \text{sp} == n )</td>
<td>( \text{sp}' == n )</td>
<td>( \pi \mapsto \Pi \mapsto \text{sp} \mapsto \text{sp}' )</td>
</tr>
<tr>
<td>( \text{sp} &lt; n )</td>
<td>( \text{sp}' &lt; n )</td>
<td>( \pi \mapsto \Pi \mapsto \text{sp} \mapsto \text{sp}' )</td>
</tr>
</tbody>
</table>

\( \text{se}_1; \text{se}_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)  \( \pi_1; \pi_2 \)

\[ \text{refining } \text{spec}([se]) \quad \text{refining } \text{spec}([e]) \quad \text{if } (\pi, \Pi) \mapsto \text{se} \subseteq e \]

| \( \text{spec} \) | \( \text{spec}([e]) \) | \( (\pi, \Pi) \mapsto \text{se} \subseteq e \) |

| \( \text{invoke}([se]) \) | \( \text{invoke}([e]) \) | \( (\pi, \Pi) \mapsto \text{se} \subseteq e \) |

| \( \text{announce}([p]) \) | \( \text{announce}([e]) \) | \( (\pi, \Pi) \mapsto \text{se} \subseteq e \) |

| \( \text{either } \{ \text{se}_1 \} \) \( \text{or } \{ \text{se}_2 \} \) | \( \text{either } \{ \text{se}_1 \} \) \( \text{or } \{ \text{se}_2 \} \) | \( (\pi, \Pi) \mapsto \text{se}_1 \subseteq e \) |

| \( \text{announce}([p]) \) | \( \text{announce}([e]) \) | \( (\pi, \Pi) \mapsto \text{se} \subseteq e \) |

| \( \text{either } \{ \text{se}_1 \} \) \( \text{or } \{ \text{se}_2 \} \) | \( \text{either } \{ \text{se}_1 \} \) \( \text{or } \{ \text{se}_2 \} \) | \( (\pi, \Pi) \mapsto \text{se}_1 \subseteq e \) |

Figure 6: Structural refinement relation (\( \subseteq \))

3.2.1 Example Handler Refinement

To illustrate the refinement rules in Figure 6, consider checking whether the handler method \( \text{update} \) 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 \( \text{se}_1; \text{se}_2 \) in Figure 6, this refinement holds if (a) \( \text{invoke}(\text{next}) \) is refined by \( \text{invoke} \) \( \text{rest} \) and (b) \( \text{establishes } \text{fe} == \text{old}(\text{fe}) \) is refined by \( \text{refining } \text{establishes } \text{fe} == \text{old}(\text{fe}) \) \( \text{Display}.\text{update}(\text{fe}); \text{fe} \).

For proving condition (a), we must check whether the subexpression \( \text{next} \) is refined by the subexpression \( \text{rest} \). This can be done by the rule for \( \text{var} \), which states that both variables \( \text{next} \) and \( \text{rest} \) must be given the same type by their respective typing contexts (\( \pi \) and \( \Pi \)). The specification typing context \( \pi \) in this case, gives type \( \text{Thunk} \) \( \text{Fig} \) to \( \text{next} \), which is the same as the type for \( \text{rest} \) given by the program typing context \( \Pi \). By applying the rule for \( \text{spec} \) in Figure 6, we can prove (b) because specification predicates \( \text{refining establishes } \text{fe} == \text{old}(\text{fe}) \) are the same in both specification expression and the program expression. Thus, the handler method \( \text{update} \) correctly refines the translucid contract for the event type \( \text{Changed} \).

The refinement rule for the case \( \text{spec} \) deserves further explanation. It states that a specification expression \( \text{spec} \) is refined by an expression \( \text{refining } \text{spec}([e]) \), which claims to refine the same specification \( \text{spec} \). The claim that \( e \) satisfies \( \text{spec} \) 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 \( \text{announce} \) and \( \text{invoke} \) 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 \( \text{announce} \) and \( \text{invoke} \) 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 \( m \) in \( \{t \text{var} \} \{e\} \) we must replace any occurrence of \( \text{announce} \) 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 \( \text{announce} \) \( \text{invoke} \) 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 \( Tr(e, \text{skip}, \bot) \). The arguments \( \text{skip} \) and \( \bot \) specify that this method does not handle any events (\( \bot \)) and thus there is no event body (\( \text{skip} \)) 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. \( \text{skip} \) is sugar for \( \text{while } 0 \{0\} \).

The result of \( Tr(e, \text{skip}, \bot) \) 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 \( t \{h \} \{t \text{var} \} \{e\} \) for each event type \( p \) with a binding when \( p \) do \( h \), one does the following. Let the contract for \( p \) be \( \text{requires } sp \) \( \text{assumes } \{se_p\} \) \( \text{ensures } sp'_p \), then compute \( Tr(e, \text{requires } sp, \text{ensures } sp'_p, p) \) 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 \( Tr(e, \text{requires } sp, \text{ensures } sp'_p, p) \) is a pure OO specification expression.
For specification expressions $se$, expressions $be$, event types $p$, where $p$ has contract requires $sp_p$ assumes $\{se_p\}$ ensures $sp'_p$ and context variables $\overline{tv}$, $Tr(se, be, p)$

<table>
<thead>
<tr>
<th>Case of $se$</th>
<th>Result</th>
<th>Side Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\varnothing$, null, new $e()$, next, $spec$</td>
<td>$se$</td>
<td>$se!=!null$ ensures $fe!=!null$</td>
</tr>
<tr>
<td>$old(se_1)$, $old(se_2)$</td>
<td>$se_1$, $se_2$</td>
<td>if $se_1 = Tr(se_1, be, p)$, $se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>$se_1$, $se_2$</td>
<td>$se_1$, $se_2$</td>
<td>$se_1 = Tr(se_1, be, p)$, $se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>either ${se_0}$ or ${se_1}$</td>
<td>either ${se_0}$ or ${se_1}$</td>
<td>if $se_0 = Tr(se_0, be, p)$, $se_1 = Tr(se_1, be, p)$</td>
</tr>
<tr>
<td>$se.m(se)$</td>
<td>$se.m(se')$</td>
<td>if $se = Tr(se, be, p)$, $se' = Tr(se', be, p)$</td>
</tr>
<tr>
<td>$se_1 = se_2$, $se_2 = se_1$</td>
<td>$se_1 = se_2$</td>
<td>$se_1 = Tr(se_1, be, p)$, $se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>if ($ep$) ${se_0}$ else ${se_1}$</td>
<td>if ($ep$) ${se_0}$ else ${se_1}$</td>
<td>if $ep = Tr(ep, be, p)$, $se_0 = Tr(se_0, be, p)$, $se_1 = Tr(se_1, be, p)$</td>
</tr>
<tr>
<td>while ($ep$) ${se}$</td>
<td>while ($ep$) ${se}$</td>
<td>if $ep = Tr(ep, be, p)$, $se' = Tr(se, be, p)$</td>
</tr>
<tr>
<td>cast $c$ $se$</td>
<td>cast $c$ $se'$</td>
<td>$se' = Tr(se, be, p)$, $se'' = Tr(se', be, p)$</td>
</tr>
<tr>
<td>$se_1$, $se_2$</td>
<td>$se_1$, $se_2$</td>
<td>$se_1 = Tr(se_1, be, p)$, $se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>$se_1 = se_2$</td>
<td>$se_1 = se_2$</td>
<td>$se_1 = Tr(se_1, be, p)$, $se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>$refining spec$</td>
<td>$spec$</td>
<td>$se_1 = Tr(se_1, be, p)$</td>
</tr>
<tr>
<td>register($se_2$)</td>
<td>$se_2$</td>
<td>$se_2 = Tr(se_2, be, p)$</td>
</tr>
<tr>
<td>invoke($se_2$)</td>
<td>$se_2$</td>
<td>$se_2 = Tr(se_2, be, p)$, $se_3 = Tr(se_3, be, p)$, $spec = requires sp_p$ ensures $sp'_p$</td>
</tr>
<tr>
<td>announce $p' {se}$</td>
<td>$se$</td>
<td>if $p'$ has transcribed contract requires $sp'<em>p$, assumes ${se</em>{p'}}$ ensures $sp''_p$ and context variables $\overline{tv}$, $se' = Tr(se, be, p)$, $se' = Tr(se', be, p)$, $se = Tr(se_1, se_2, p')$, $spec = requires sp'_p$ ensures $sp''_p$</td>
</tr>
</tbody>
</table>

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 $Tr(se, be, p)$, with $p$ as the announced event and $be$ 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 either – 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 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 either – or choice expression guarded by a 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 invoke ($se_1$), the either-branch contains a sequence of two expressions: translation of the argument $se_1$ and the event body $be$, which means no more handler to run. The or-branch contains a sequence of two expressions: translation of argument $se_1$ and translation of the translucid contract $sp_p$. The guarding refining expression assures that specification $spec$ is satisfied by the choice expression inside. $spec$ contains pre- and post-condition of the contract $se_{p'}$.

Translation of announce expression is similar to the invoke. In case of announce $p' \{se\}$, the either-branch contains a sequence of two expressions: translation of the argument $se$ and the translation of event body $se_1$. In or-branch the first expression is the translation of the arguments and their assignment to context variables $\overline{tv'}$. 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 $Tr(se, skip)$ with $se = announce$ Changed(this) \{ this.x = x; 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.

![Figure 9: Translation of method setX](image-url)

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 $announce$ and $invoke$ expressions to a problem similar to reasoning about loops. Thus, standard techniques for reasoning about
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 checks 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 \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 the 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. After-augmentation handler is executed after the event body whereas in the before augmentation the order is opposite.

![Figure 10: Unrolling translation function](image1)

![Figure 11: Specifying augmentation with a translucid contract](image2)

![Figure 12: Specifying augmentation with a translucid contract](image3)

![Figure 13: After-augmentation handler](image4)

Structural similarity requires the handler implementation to evaluate \texttt{invoke} exactly once and at its very beginning which in turn
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.

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).

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 }

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 }

27 class Unique{
28  16 HashMap colors;
29  17 Color check (thunk Color rest, Fig fe){
30    while(fe.colFix == 0){
31      invoke(next);
32      fe.colFix = 1;
33   }
34   else
35     refining preserves fe.colFix == 0  
36   }
37 }
38 }
39  }
40 }
41 }

42 Color event ClChanged{
43  Fig fe;
44  requires fe != null
45  assumes {
46   while(fe.colFix == 0){
47     invoke(next);
48     fe.colFix = 1;
49   }
50   while(fe.colFix == 0){
51     invoke(next);
52   }
53   }
54  }
55  }
56  }
57 }
58 }
59 }

Figure 16: Specifying replacement with a transluic contract

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 vi-
olated. 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
functionality, can evaluate invoke expression any number of
times. Figure 18 illustrates a combination contract and a handler.
The transluclid contract in the event type specification on lines 5–11
allows an invoke expression to be evaluated zero or more num-
ber 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
crude 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
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 al-
ready 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 com-
bination 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 ex-
pression refining the either-block or the or-branch in the specifi-
cation. 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 be-
longs 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 aug-
mentation, 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 dis-
tinguished 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
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) s 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). However translucid contracts can. On lines 6–9 there is a specification that conveys to the developer of the class Fig 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 crossing-cutting. It relies on textual copying of specifications for specification inheritance and syntactical weaving of specification for specification crossing-cutting. 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 Steinmann et al.’s join point types [26] share similar spirit with Rajan and Levan’s event types [19]. Although Steinmann 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 verification solves the problem of reasoning about 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