8-4-2014

Modular Reasoning in the Presence of Event Subtyping

Mehdi Bagherzadeh
Iowa State University, mbagherz@iastate.edu

Robert Dyer
Bowling Green State University, rdyer@bgsu.edu

Rex D. Fernando
University of Wisconsin - Madison, rex@cs.wisc.edu

Hridesh Rajan
Iowa State University, hridesh@iastate.edu

Jose Sanchez
University of Central Florida, sanchez@eecs.ucf.edu

Modular Reasoning in the Presence of Event Subtyping

Abstract

Separating crosscutting concerns while preserving modular reasoning is challenging. Type-based interfaces (event types) separate modularized crosscutting concerns (observers) and traditional object-oriented concerns (subjects). Event types paired with event specifications have been shown to be effective in enabling modular reasoning about subjects and observers. Similar to class subtyping there are benefits to organizing event types into subtyping hierarchies. However, unrelated behaviors of observers and their arbitrary execution orders could cause unique, somewhat counterintuitive, reasoning challenges in the presence of event subtyping. These challenges threaten both tractability of reasoning and reuse of event types. This work makes three contributions. First, we pose and explain these challenges. Second, we propose an event-based calculus to show how these challenges can be overcome. Finally, we present modular reasoning rules of our technique, and show its applicability to other event-based techniques including join point types.

Keywords

event type inheritance, modular reasoning about behaviors and control effects, refinement of event specifications, subject, observer

Disciplines

Programming Languages and Compilers

Modular Reasoning in the Presence of Event Subtyping

Mehdi Bagherzadeh[α] Robert Dyer[β] Rex D. Fernando[γ] Hridesh Rajan[α] José Sánchez[θ]

TR #14-02b
Initial Submission: December 11, 2013
Revised: Ocrober 10, 2014, August 04, 2014, March 25, 2014

Keywords: event type inheritance, event subtyping, modular reasoning about behaviors and control effects, refinement of event specifications, subject, observer

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

Department of Computer Science
226 Atanasoff Hall
Iowa State University
Ames, Iowa 50011-1041, USA

Modular Reasoning in the Presence of Event Subtyping

Mehdi Bagherzadeh[α] Robert Dyer[β] Rex D. Fernando[γ] Hridesh Rajan[α] José Sánchez[θ]

[α]Iowa State University [β]Bowling Green State University [γ]University of Wisconsin, Madison [θ]University of Central Florida

[α]{mbagherz,hridesh}@iastate.edu [β]rdyer@bgsu.edu [γ]rex@cs.wisc.edu [θ]sanchez@eecs.ucf.edu

Abstract

Separating crosscutting concerns while preserving modular reasoning is challenging. Type-based interfaces (event types) separate modularized crosscutting concerns (observers) and traditional object-oriented concerns (subjects). Event types paired with event specifications were shown to be effective in enabling modular reasoning about subjects and observers. Similar to class subtyping, organizing event types into subtyping hierarchies is beneficial. However, unrelated behaviors of observers and their arbitrary execution orders could cause unique, somewhat counterintuitive, reasoning challenges in the presence of event subtyping. These challenges threaten both tractability of reasoning and reuse of event types. This work makes three contributions. First, we pose and explain these challenges. Second, we propose an event-based calculus to show how these challenges can be overcome. Finally, we present modular reasoning rules of our technique and show its applicability to other event-based techniques.

1. Introduction

Separation of crosscutting concerns has generated significant interest over the past decade or so [1–15]. An interesting challenge in separation of crosscutting concerns is to preserve modular reasoning and its underlying modular type checking. Recently some consensus has been formed that a notion of explicit interfaces, between modularized crosscutting concerns and traditional object-oriented (OO) concerns, enables modular type checking [8–15] and modular reasoning [2–12].

Previous work, such as join point types (JPT) [15], join point interfaces (JPI) [14], and Ptolemy's typed events [16], just to name a few, propose a type-based formulation of these interfaces to enable modular type checking. These type-based interfaces could be thought of as *event types* which are announced, implicitly or explicitly, by traditional OO concerns, or *subjects*, where modularized crosscutting concerns, or *observers*, register for the events and run upon their announcement [17, 18]. Announcement of an event type could cause *zero or more* of its observers to run in a chain where observers can invoke each other. This event announcement and handling model for separation of concerns has been popularized by AspectJ [1] and is different from models in which the subject is responsible for invoking all of its observers, as in Java's event model and the Observer pattern.

Similar to OO subtyping, where a class can subtype another class, an event type can subtype another event type. *Event subtyping* enables structuring of event types and allows for code reuse [14–16]. Code reuse allows *an observer of an event to run upon announcement of any of its subevents*, i.e. observer reuse, and makes the data attributes of the event accessible in its subevents, i.e. event inheritance. Previous work [14–16] enables modular type checking of subjects and observers in the presence of event subtyping.

Modular reasoning about subjects and observers, unlike their modular type checking, is focused on understanding their behaviors [3, 19], control effects [5, 7, 20], data effects [2, 21], and exception flows [6]. In modular reasoning, a system is understood one module at a time and in isolation, using only its implementation and the interfaces, not implementations, of other modules it references [10, 11]. Previous work, such as crosscutting programming interfaces (XPI) [3], crosscutting programming interfaces with design rules (XPIDR) [20] and translucid contracts [5–7], enables modular reasoning about subjects and observers using *event specifications*, however, they do not support event subtyping.

Modular reasoning about behaviors of subjects and observers, using event specifications of event types that can subtype each other, where announcement of an event allows not only observers of the event but also observers of *all* of its superevents, with possibly *unrelated behaviors*, run in an *arbitrary order*, faces the following unique challenges:

- *Problem (1) – Combinatorial reasoning*: unrelated behaviors of observers may require a factorial number of combinations of execution orders of observers of the event and observers of all of its superevents, up to $n!$ for n observers, to be considered in reasoning about the subject, which makes reasoning intractable;
- *Problem (2) – Behavior invariance*: arbitrary execution orders of observers may force observers of the event and observers of all of its superevents to satisfy the same behavior, which prevents reuse of event types, their specifications and observers.

In this work, we solve problem *(1)* by imposing a novel *refining relation* among specifications of an event and its superevents, such that for each event in a subtyping hierarchy, its greybox specification [22] refines both behaviors and control effects of the greybox specification of its superevent. Our refining relation is the inverse of the classical refining for blackbox specifications [23] and extends it to greybox specifications with control effect specifications. We solve problem *(2)* by imposing a *non-decreasing relation* on execution orders of observers of an event and observers of its superevents, such that for each event in a subtyping hierarchy, observers of an event run before observers of its superevents. With the refining and non-decreasing relations combined, subjects and observers of an event could be understood modularly and in a tractable manner using only the specification of their event, independent of observers of the event, observers of its superevents and their execution orders, while allowing reuse. This is only sound when we impose a *conformance relation* on subjects and observers of an event such that each subject and observer of the event respects behaviors and control effects of their event specifications.

We illustrate problems *(1)–(2)* in the event-based language Ptolemy [16] by adding greybox event specifications to it, and propose our solution in the context of a new language design called *Ptolemy*$_{\mathbb{S}}$. The language *Ptolemy*$_{\mathbb{S}}$ has built-in support for the refining, non-decreasing and conformance relations that together enable modular reasoning about behaviors and control effects of subjects and observers. Our proposed solution could be applied to other event-based systems especially those with event announcement and handling models similar to AspectJ [1], including join point types [15] and join point interfaces [14].

Contributions We make the following contributions:

- identification and illustration of problems *(1)–(2)* of modular reasoning about subjects and observers, in the presence of event subtyping;
- the refining relation for greybox event specifications, the non-decreasing relation for execution orders of observers and the conformance relation for behaviors and control effects of subjects and observers of an event hierarchy, to solve problems *(1)–(2)* and enable modular reasoning;
- *Ptolemy*$_{\mathbb{S}}$, a language design and its sound semantics with support for the refining, non-decreasing and conformance relations;
- *Ptolemy*$_{\mathbb{S}}$'s Hoare logic [24] for modular reasoning; and
- applicability of *Ptolemy*$_{\mathbb{S}}$'s reasoning to AspectJ-like event-based systems including join point types [15].

Implementation of *Ptolemy*$_{\mathbb{S}}$'s compiler is publicly available at `http://sourceforge.net/p/ptolemyj/code/HEAD/tree/pyc/branches/event-inheritance/`.

Outline Section 2 illustrates problems *(1)–(2)* of modular reasoning about subjects and observers using Ptolemy [16] and translucid contracts [5]. Section 3 discusses the refining and non-decreasing relations in the context of *Ptolemy*$_{\mathbb{S}}$. Section 4 discusses modular reasoning using translucid contracts, and its soundness using the conformance relation. Section 5 shows the applicability of *Ptolemy*$_{\mathbb{S}}$'s reasoning technique to join point types [15]. Section 6 illustrates *Ptolemy*$_{\mathbb{S}}$'s modular reasoning about control effects and subject-observer control interference [25]. Section 7 discusses the implementation and limitations of the approach. Section 8 presents related work and Section 9 discusses future work and concludes.

Appendix Sections A and B discuss *Ptolemy*$_{\mathbb{S}}$'s static and dynamic semantics. Proofs for soundness of *Ptolemy*$_{\mathbb{S}}$'s Hoare logic and soundness of type system along with other details of *Ptolemy*$_{\mathbb{S}}$ can also be found in appendices.

2. Problems

In this section we illustrate problems *(1)–(2)*, discussed in Section 1, using the event-based language Ptolemy [16].

As an example of modular reasoning about the behavior of a subject, consider *static* verification of the JML-like assertion Φ on line 8 of Figure 1. The assertion says that: *the expression* `e` *and its state remain the same after announcement and handling of the event type* `AndEv`*, on lines 4–7*, where `AndEv` is a subevent of `BinEv` and `ExpEv`, in the event subtyping hierarchy of Figure 2. The assertion assumes that `e`, `e.left`, and `e.right` are not null. The method `equals` checks for equality of two objects and their states, e.g. two expressions of type `AndExp` are equal, if their object references, `parent`s and their `left` and `right` children are equal. The expression **`old`** refers to values of variables at the beginning

```
1  /* subject */
2  class ASTVisitor {
3   void visit(AndExp e) {
4    announce AndEv(e, e.left, e.right) {
5     e.left.accept(this);
6     e.right.accept(this);
7    }
8    assert e.equals(old(e)); Φ
9   }
10  void visit(TrueExp e) { announce TrueEv(e) {} } ..
11 }
```

Figure 1. Static verification of Φ in subject `ASTAnnouncer`.

```
12 /* event types */
13 void event ExpEv                  { Exp node; }
14 void event BinEv  extends ExpEv {
15  BinExp node; Exp left, right;
16 }
17 void event AndEv  extends BinEv { AndExp node; }
18 void event UnEv   extends ExpEv { UnExp node; }
19 void event TrueEv extends UnEv  { TrueExp node; }
20 /* data types */
21 class Exp {
22  Exp parent;
23  void accept(ASTVisitor v) { v.visit(this); }
24 }
25 class BinExp extends Exp    { Exp left, right; .. }
26 class AndExp extends BinExp { .. }
27 class UnExp  extends Exp    { .. }
28 class TruExp extends UnExp  { .. }
```

Figure 2. Event `AndEv` and its superevents `BinEv` and `ExpEv`.

of method `visit`, on line 3. To better understand the problems of modular reasoning we first provide a short background on Ptolemy.

2.1 Ptolemy in a Nutshell

Ptolemy [16] is an extension of Java for separation of crosscutting concerns [13] with support for event types, event subtyping and explicit announcement and handling of events. In Ptolemy, a subject announces an event and observers register for the event and run upon its announcement. Announcement of an event causes observers of the event and observers of its superevents to run in a chain according to their *dynamic registration order* where observers can invoke each other.

Written in Ptolemy, Figures 1, 2 and 3 together show a simple expression language with a tracer, type checker and evaluator for boolean expressions such as `AndExp`, `OrExp` and numerical expressions. We focus on the code for boolean expressions, however, the complete code can be found at `http://sf.net/p/ptolemyj/code/HEAD/tree/pyc/branches/event-inheritance/examples/100-Polymorphic-Expressions`. A parser generates abstract syntax trees (AST) for expressions of the language and provides a visitor to visit these abstract syntax trees.

The subject `ASTVisitor`, in Figure 1, uses **`announce`** expressions to announce event types for each node type in the AST of an expression, upon its visit. For example, it announces the event type `AndEv` for visiting `AndExp`, on lines 4–7, with its event body on lines 5–6. Observers `Tracer`, `Checker`, and `Evaluator`, in Figure 3, show interest in events and register to run upon their announcement. For example, `Evaluator` shows interest in `AndEv`, using a **`when`** – **`do`** binding declaration, on line 64, and registers for it using a **`register`** expression, on line 57. `Evaluator` runs the observer handler method[1] `evalAndExp`, on lines 58–63, upon announcement of `AndEv`. The handler pops up values of the left and

[1] Phrases 'observer' and 'observer handler method' are used interchangably.

```
30 /* observers */
31 class Tracer {
32  Tracer() { register(this); }
33  void printExp(ExpEv next) {
34   next.invoke();
35   logVisitEnd(next.node());
36  }
37  when ExpEv do printExp;
38 }
39 class Checker{
40  Stack<Type> typeStack = ..
41  Checker() { register(this); }
42  void checkBinExp (BinEv next) {
43   next.invoke();
44   Bool t1 = (Bool) typeStack.pop();
45   Bool t2 = (Bool) typeStack.pop();
46   typeStack.push(new Bool());
47  }
48  when BinEv do checkBinExp;
49  void checkUnExp(UnEv next) {
50   next.invoke();
51   typeStack.push(new Bool());
52  }
53  when UnEv do checkUnExp;
54 }
55 class Evaluator {
56  Stack<Value> valStack = ..
57  Evaluator() { register(this); }
58  void evalAndExp (AndEv next) {
59   next.invoke();
60   BoolVal b1 = (BoolVal) valStack.pop();
61   BoolVal b2 = (BoolVal) valStack.pop();
62   valStack.push(new BoolVal(b1.val && b2.val));
63  }
64  when AndEv do evalAndExp;
65  void evalTrueExp (TrueEv next) {
66   next.invoke();
67   valStack.push(new BoolVal(true));
68  }
69  when TrueEv do evalTrueExp; ..
70 }
```

Figure 3. Observers `Tracer`, `Checker` and `Evaluator`.

right children of the visited `AndExp` node from a value stack, conjoins them together to evaluate the value of the conjunct expression and pushes the result back to the stack. For a binary boolean expression, `Checker` ensures that its children are boolean expressions by popping and casting their boolean values from a type stack. Types `Type` and `Value` and their subtypes, e.g. `Bool` and `BoolVal`, denote types and values of boolean and numerical expressions.

Announcement of `AndEv`, on lines 4–7, could cause the observer `Evaluator` of the event, and observers `Checker` and `Tracer` of its superevents `BinEv` and `ExpEv` to run in a chain, if they are registered. An observer of an event is bound to the event through a binding declaration. For example, `Evaluator` is an observer of `AndEv` because of its binding declaration whereas `Checker` is not, though it may run upon announcement of `AndEv`. Observers are put in a chain of observers as they register for an event with the event body as the last observer. For example, the event body for `AndEv` is the last observer of the event in the chain. The chain of observers is stored inside an event closure represented by a variable **`next`** and the chain is passed to each observer handler method. For example, the chain is passed to `evalAndExp` on line 58. An observer of an event can invoke the next observer in the chain using an **`invoke`** expression which is similar to AspectJ's **`proceed`**. Dynamic registration of observers allows observers to register in any arbitrary order which in turn means that an observer of an event can invoke another observer of the same event, an observer of any of its superevents, or any of its subevents. For example, the observer `Evaluator` for the event `AndEv` can invoke, on line 59, another observer of `AndEv` or any of its superevents or subevents.

Event types must be declared before they are announced by subjects or handled by observers. An event declaration names a superevent in its **`extends`** clause and a set of context variables in its body. Context variables are shared data between subjects and observers of an event. An event inherits contexts of its superevents via event inheritance, can redeclare contexts of its superevents via depth subtyping, or add to them via width subtyping. For example, the declaration of `AndEv` extends `BinEv` as its superevent, inherits its context variables `left` and `right` and redeclares its context `node`. The declaration of `BinEv`, on lines 14–16, adds contexts `left` and `right`, using width subtyping, to `node` that it inherits from its superevent `ExpEv`. Contexts `left` and `right` serve illustration purposes only, otherwise they could be projected from `node`. Values of context variables of an event are set upon its announcement and stored in its event closure. For example the contexts `node`, `left` and `right` of `AndEv` are set with values `e`, `e.left` and `e.right` upon announcement of `AndEv`, on line 4.

2.1.1 Event Type Specifications

To verify Φ in Figure 1, the behavior of the announce expression for `AndEv`, on lines 4–7, must be understood, which in turn is dependent on behaviors of observers of `AndEv` and observers of its superevents, running upon its announcement. For such understanding to be modular, only the implementation of the subject `ASTVisitor`, on lines 2–11, and interfaces of modules it references, including the event types `AndEv` and its superevents `BinEv` and `ExpEv`, are available. However, neither `ASTVisitor` nor `AndEv`, `BinEv`, `ExpEv` say anything about the behaviors of their observers, which in turn makes modular verification of Φ difficult.

Previous work [5–7] proposes translucid contracts as event type specifications to specify behaviors and control effects of subjects and observers of an event and enables their modular reasoning in the *absence* of event subtyping. We add translucid contracts to Ptolemy's event types and illustrate how unrelated event specifications of events in a subtyping hierarchy and arbitrary execution of their observers could cause problems *(1)–(2)* in modular reasoning about subjects and observers in the *presence* of event subtyping.

```
1 void event ExpEv { ..
2  requires node != null
3  assumes {
4   next.invoke();
5   requires true
6   ensures next.node().parent == old(next.node().parent);
7  }
8  ensures node.equals(old(node))
9 }
10 void event BinEv extends ExpEv { ..
11  requires left != null && right != null && node != null
12  assumes {
13   next.invoke();
14   requires next.node().left!=null&&next.node().right!=null
15   ensures next.node().parent == old(next.node().parent);
16  }
17 ensures true
18 }
19 void event AndEv extends BinEv { ..
20  requires left != null && right != null && node != null
21  assumes {
22   next.invoke();
23   requires next.node().left!=null&&next.node().right!=null
24   ensures next.node().parent == old(next.node().parent);
25  }
26  ensures node.equals(old(node))
27 }
```

Figure 4. Unrelated contracts of subtyping events.

In its original form [5], a translucid contract of an event is a greybox specification [22] that specifies behaviors and control effects of individual observers of the event, with *no relation* to behaviors and control effects of its superevents or subevents. Figure 4 shows translucid contracts of a few event types of Figure 2. The translucid contract of `AndEv`, on lines 20–26, specifies behavior and control effects of the observer `Evaluator` of `AndEv` and especially its observer handler method `evalAndExp`. The behavior of `evalAndExp` is specified using the precondition **requires**, on line 20, and the postcondition **ensures**, on line 26, which says that the execution of the observer starts in a state in which the context `node`, `left` and `right` are not null, i.e. $left != \textbf{null}$ && $right != \textbf{null}$ && $node != \textbf{null}$, and if the execution terminates, it terminates in a state in which the node is the same as before the start of the execution of the observer, i.e. $node.equals(\textbf{old}\,(node))$.

Control effects of `evalAndExp` are specified by the **assumes** block, on lines 21–25, that limits its implementation structure. The assumes block is a combination of program and specification expressions. The program expression $\textbf{next}.\textbf{invoke}()$, on line 22, specifies and exposes control effects of interest, e.g. occurrence of the invoke expression in the implementation of `evalAndExp`, and the specification expression $\textbf{requires}\ \textbf{next}.node().left != \textbf{null}$ && $\textbf{next}.node().right != \textbf{null}$ $\textbf{ensures}\ \textbf{next}.node().parent == \textbf{old}\,(\textbf{next}.node().parent)$, on lines 23–24, hides the rest of the implementation of `evalAndExp`, allowing it to vary as long as it respects the specification. The assumes block of `AndEv` says that an observer `evalAndExp` of `AndEv` must invoke the next observer in the chain of observers, line 22, and then can do anything as long as it does not modify the `parent` field of the context variable `node`, line 23–24. The expression $\textbf{next}.node()$ in the contract retrieves the context `node` from the event closure next for `AndEv` and the expression **old** refers to values of variables before event announcement.

Through the specification of behaviors of observers of an event, the translucid contract of an event also specifies the behavior of an invoke expression in the implementation of an observer of the event. This is true because in the absence of event subtyping the invoke expression causes the invocation of the next observer of the same event. For example, the contract of `AndEv` specifies the behavior of the invoke expression in the implementation of the observer handler method `evalAndExp` to have the precondition $left != \textbf{null}$ && $right != \textbf{null}$ && $node != \textbf{null}$ and the postcondition $node.equals(\textbf{old}\,(node))$. The precondition of the invoke expression must hold right before its invocation and its postcondition must hold right after it.

2.2 Combinatorial Reasoning, Problem (1)

Various execution orders of observers of an event and observers of its superevents could yield different behaviors, especially if there is no relation between behaviors of observers of the event and its superevents and no known order on their execution. Combinatorial reasoning forces all such variations of execution orders to be considered in reasoning about a subject of an event, which makes the reasoning intractable [18].

To illustrate, reconsider static verification of Φ for announcement of `AndEv`, on lines 4–7 of Figure 1, with an observer instance `evaluator` registered to handle `AndEv` and an observer instance `checker` registered to handle `BinEv`. Translucid contracts of `AndEv` and `BinEv` in Figure 4 specify the behaviors of `evaluator` and `checker`, respectively. Announcement of `AndEv` could cause the observers `evaluator` and `checker` to run in two alternative execution orders χ_1: `evaluator` $\rightharpoonup$ `checker` or χ_2: `checker` $\rightharpoonup$ `evaluator`, depending on their dynamic registration order. In χ_1, `evaluator` runs first, where it invokes `checker` using its invoke expression, on line 59 of Figure 3, and the opposite happens in χ_2. Body of `AndEv` runs as the last observer in χ_1 and χ_2 (not shown).

For χ_1, the assertion Φ could be verified using the contract of `AndEv` for `evaluator`, on lines 20–26 of Figure 4, using its postcondition $node.equals(\textbf{old}\,(node))$, on line 26. Recall that the precondition and postcondition of `AndEv` are the precondition and postcondition of its observer `evaluator`. To verify Φ, the postcondition of `AndEv` is copied right after the announce expression, using the copy rule [27], and its context variables `node`, `left` and `right` are replaced respectively with parameters `e`, `e.left` and `e.right` of the announce expression [5]. This allows use of the postcondition of the contract of `AndEv` in the scope of the method `visit`. Replacing the context variables in the postcondition of `AndEv` produces the predicate $e.equals(\textbf{old}\,(e))$ which is exactly the assertion Φ that we wanted to prove.

In χ_1 the assertion Φ could be verified using the postcondition of the translucid contract of `AndEv` alone. An example of a more subtle interplay of behaviors of `evaluator` and `checker` is a scenario in which translucid contracts of `AndEv` and `BinEv` look like $\textbf{requires true assumes}\,\{\ \textbf{establishes true};\ \textbf{next}.\textbf{invoke}();\}\ \textbf{ensures true}$ and $\textbf{requires true assumes}\ \{\textbf{establishes}\ node.equals\ (\textbf{old}\,(node));\ \textbf{next}.\textbf{invoke}();\}\ \textbf{ensures true}$, respectively. The specification expression $\textbf{establishes}\ ep$ is a sugar for $\textbf{requires true ensures}\ ep$. With these contracts, neither the postcondition of `AndEv` nor `BinEv` alone are enough to verify Φ but their interplay results in a postcondition that implies and consequently verifies Φ.

In contrast, Φ cannot be statically verified for χ_2 because neither the postcondition **true** of the contract of `BinEv`, on line 17 of Figure 4, nor the interplay of behaviors of observers `evaluator` and `checker` in χ_2 provides the guarantees required by Φ.

As illustrated, in reasoning about a subject of an event, various execution orders of its observers and observers of its superevents must be considered. Generally for *n* observers of events in a subtyping hierarchy there can be up to *n*! possible execution orders [6, 18] which in turn makes the reasoning intractable. Also dependency of the reasoning on execution orders of observers *threatens the modularity* of the reasoning. This is because any changes in execution orders of observers could invalidate any previous reasoning. For example the already verified assertion Φ for the execution order χ_1 is invalidated by changing the execution order to χ_2.

2.3 Behavior Invariance, Problem (2)

In reasoning about an observer of an event, arbitrary execution orders of observers of the event and observers of its superevents in a chain, could force observers of the event and observers of all of its superevents in a subtyping hierarchy to satisfy the same behavior. This could prevent reuse of event types, their specifications [28] and their observers [14, 15].

To illustrate, consider reasoning about the behavior of the invoke expression in the observer `evaluator`, in Figure 3 line 59, with an observer instance `evaluator` registered to handle `AndEv` and observer instance `tracer` registered to handle its transitive superevent `ExpEv`. Translucid contracts of `AndEv` and `ExpEv` in Figure 4 specify behaviors of `evaluator` and `tracer`, respectively. Upon announcement of `AndEv`, observers `evaluator` and `tracer` could run in two alternative execution orders of χ_1: `evaluator` $\rightharpoonup$ `tracer` or χ_2: `tracer` $\rightharpoonup$ `evaluator`.

Recall that the translucid contract of an event also specifies behaviors of invoke expressions in implementations of its observers. In other words, the contract of `AndEv` specifies the behavior of the invoke expression in its observer `evaluator`, on line 59. That is, the precondition $left != \textbf{null}$ && $right != \textbf{null}$ && $node != \textbf{null}$ of `AndEv` must hold right before the invoke expression in `evaluator` and the postcondition $node.equals(\textbf{old}\,(node))$ must hold right after the invoke expression.

In χ_1, for the invoke expression of `evaluator` to invoke `tracer`, its precondition must imply the precondition $node != \textbf{null}$ of `tracer` and the postcondition $node.equals(\textbf{old}\,(node))$

of `tracer` must imply the postcondition of the invoke expression in `evaluator`. In other words, χ_1 requires $\omega_1 : \mathscr{P}(\texttt{AndEv}) \Rightarrow \mathscr{P}(\texttt{ExpEv}) \wedge \mathscr{Q}(\texttt{ExpEv}) \Rightarrow \mathscr{Q}(\texttt{AndEv})$ to hold for `evaluator` to invoke `tracer`. Auxiliary functions $\mathscr{P}$ and $\mathscr{Q}$ return the precondition and postcondition of an event type, respectively. In contrast, χ_2 requires $\omega_2 : \mathscr{P}(\texttt{ExpEv}) \Rightarrow \mathscr{P}(\texttt{AndEv}) \wedge \mathscr{Q}(\texttt{AndEv}) \Rightarrow \mathscr{Q}(\texttt{ExpEv})$ to hold for `tracer` to invoke `evaluator`. To allow both execution orders χ_1 and χ_2, both conditions ω_1 and ω_2 must hold which in turn requires preconditions and postconditions of `AndEv` and `ExpEv` and consequently preconditions and postconditions of their observers `evaluator` and `tracer` to be the same, i.e. invariant.

3. Solution

To solve combinatorial reasoning and behavior invariance problems we propose to *(1)* relate behaviors of observers of an event and its superevent by a refining relation among greybox event specifications in an event subtyping hierarchy and *(2)* to limit arbitrary execution order of observers by a non-decreasing relation on execution orders of observers. This proposal constitutes a new language design called *Ptolemy*$_{\mathbb{S}}$ with support for these relations. Figure 5 shows an overview of these relations in *Ptolemy*$_{\mathbb{S}}$.

In Figure 5, for an event subtyping hierarchy, the refining relation guarantees that the specification (contract) of an event refines the specification of its superevent and the non-decreasing relation guarantees that upon announcement of an event by a subject, an observer of the event runs before an observer of its superevent. The conformance relation guarantees that each subject and observer of an event conform to and respect their event specification.

Detailed formalization of *Ptolemy*$_{\mathbb{S}}$'s sound static and dynamic semantics can be found in Sections B and A.

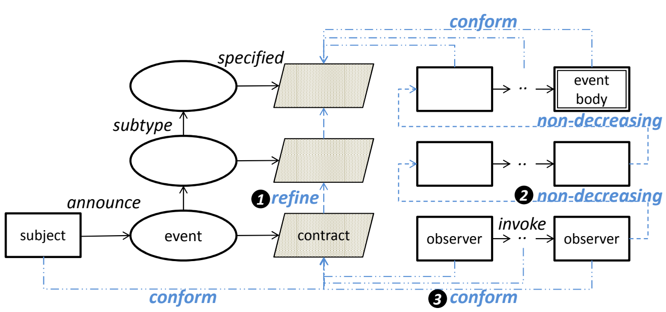

Figure 5. Refining, non-decreasing and conformance relations.

3.1 *Ptolemy*$_{\mathbb{S}}$'s Syntax

Figure 6 shows the expression-based core syntax of *Ptolemy*$_{\mathbb{S}}$ with focus on event types, event subtyping, and event specifications. Hereafter, $term^*$ means a sequence of zero or more terms and $[term]$ means zero or one term.

A *Ptolemy*$_{\mathbb{S}}$ program is a set of declarations followed by an expression, which is like a call to the main method in Java. There are two kinds of declarations: class and event type declarations. A class can extend another class and it may have zero or more fields, methods, and binding declarations.

Similarly, an event type declaration can extend (subtype) another event type and has a return type, a set of context variable declarations, and an optional translucid contract. The return type of an event specifies the return type of its observers. An interesting property of return types of subtyping events is that, because of the non-decreasing relation, the return type of an event is a supertype of the return type of the event it extends, see Section B. An event type declaration inherits context variables of the event types it extends and can declare more through width subtyping. It can also

```
prog ::= decl* e
decl ::= class c extends d { form* meth* binding* }
       | c event ev extends ev' { form* [contract] }
meth ::= t m (form*) { e }
binding ::= when ev do m
e, se ::= var | null | new c () | cast c e
       | e.m(e*) | e.f | e.f = e | form = e ; e
       | announce ev (e*) { e } | e.invoke()
       | register(e) | unregister(e)
       | refining spec { e } | spec | either {e} or {e}
p, q ::= var | p.f | p == p | p < p | ! p | p && p | old(p)
contract ::= requires p [assumes { se }] ensures q
spec ::= requires p ensures q
t ::= c | thunk ev
form ::= t var
```

$c,\ d$	$\in \mathscr{C} \cup \{\texttt{Object}\}$	set of class names
$ev,\ ev'$	$\in \mathscr{E} \cup \{\texttt{Event}\}$	set of event names
f	$\in \mathscr{F}$	set of field names
var	$\in \mathscr{V} \cup \{\texttt{this}, \texttt{next}\}$	$\mathscr{V}$ is a set of variable names

Figure 6. *Ptolemy*$_{\mathbb{S}}$'s core syntax, based on [5, 13, 16].

redeclare the context variables of the event types it extends through depth subtyping [16] as long as the type of the redeclaring context is a subtype of the type of the redeclared context. Figure 2 illustrates the declaration of the event type `AndEv`, on line 17.

3.2 Refining Relation of Event Specifications

Ptolemy$_{\mathbb{S}}$ relates behaviors and control effects of observers of events in a subtyping hierarchy by relating their greybox event specifications through a refinement relation $\trianglelefteq$. In the refining relation the specification of an event refines the specification of its superevent, for both behaviors and control effects. *Ptolemy*$_{\mathbb{S}}$'s refinement among greybox event specifications is the inverse of classical behavioral subtyping for blackbox method specifications [23], however, blackbox specifications do not specify control effects.

In *Ptolemy*$_{\mathbb{S}}$, a translucid contract [5, 6] of an event is a greybox specification that, *in relation* to its superevents, specifies behaviors and control effects of individual observers of the event and their invoke expressions. A translucid contract of an event specifies behaviors using the precondition **`requires`** and the postcondition **`ensures`**. The behavior **`requires`** p **`ensures`** q says that if the execution of an observer of the event starts in state σ satisfying p, written as $\sigma \models p$, and it terminates normally, it terminates in a state σ' that satisfies q, i.e. $\sigma' \models q$.

A translucid contract specifies control effects of its individual observers using its **`assumes`** block. An assumes block is a combination of program and specification expressions. A program expression exposes control effects of interest, e.g. invoke expressions, in the implementation of an observer whereas a specification expression *spec* hides the rest of its implementation allowing it to vary as long it respects its specification. The contract of an event only names the context variables of the event and must expose invoke expressions in the implementation of its observers. Figure 4 illustrates the translucid contract of `AndEv`, on lines 20–26, with its precondition, on line 20, postcondition, on line 26, program expression, on line 22 and specification expression, on lines 23–24. *Ptolemy*$_{\mathbb{S}}$ relates translucid contracts of an event and its superevents through the refining relation $\trianglelefteq$.

DEFINITION 3.1. ***(refining translucid contracts)**. For event types ev and ev', where ev is a subevent of ev', written as $ev \ll: ev'$*[2]*, and their respective translucid contracts $\mathscr{G} = ($* **`requires`** *p* **`assumes`** *$\{se\}$* **`ensures`** *$q)$ and $\mathscr{G}' = ($* **`requires`** *p'*

[2] The class subtyping relation $\preccurlyeq$ is different from *Ptolemy*$_{\mathbb{S}}$'s event subtyping relation $\ll:$, discussed in Section B.

assumes $\{se'\}$ ***ensures*** q'), $\mathscr{G}'$ *is refined by* $\mathscr{G}$*, written as* $\mathscr{G}' \trianglelefteq \mathscr{G}$*, if and only if:*

(i). $\texttt{requires}\ p'\ \texttt{ensures}\ q' \trianglelefteq \texttt{requires}\ p\ \texttt{ensures}\ q$

(ii). $se' \trianglelefteq se$

Figure 7 defines the refinement relation $\trianglelefteq$ *for Ptolemy*$_{\mathbb{S}}$ *expressions.*

Event specification refinement relation: $\boxed{\Gamma \vdash se' \trianglelefteq se}$

(R-SPEC)
$$\frac{spec = \texttt{requires}\ p\ \texttt{ensures}\ q \quad spec' = \texttt{requires}\ p'\ \texttt{ensures}\ q' \quad p \Rightarrow p' \quad q' \Rightarrow q}{\Gamma \vdash spec' \trianglelefteq spec}$$

(R-INVOKE)
$$\frac{\Gamma \vdash se' \trianglelefteq se}{\Gamma \vdash se'.\texttt{invoke}() \trianglelefteq se.\texttt{invoke}()}$$

(R-VAR)
$$\frac{textualMatch(var', var)}{\Gamma \vdash var' \trianglelefteq var}$$

(R-DEFINE)
$$\frac{\Gamma \vdash se'_1 \trianglelefteq se_1 \quad \Gamma, t : var \vdash se'_2 \trianglelefteq se_2}{\Gamma \vdash t\ var = se'_1; se'_2 \trianglelefteq t\ var = se_1; se_2}$$

(R-IF)
$$\frac{\Gamma \vdash sp' \trianglelefteq sp \quad \Gamma \vdash se'_1 \trianglelefteq se_1 \quad \Gamma \vdash se'_2 \trianglelefteq se_2}{\Gamma \vdash \texttt{if}(sp')\{se'_1\}\ \texttt{else}\{se'_2\} \trianglelefteq \texttt{if}(sp)\{se_1\}\ \texttt{else}\{se_2\}}$$

Figure 7. Select rules for the refining relation $\trianglelefteq$.

In Definition 3.1, for a translucid contract of an event to refine the contract of its superevent, *(i)* its behavior must refine the behavior of the contract of the superevent and *(ii)* its assumes block must refine the assumes block of the translucid contract of its superevent.

In Figure 7, the rule (R-SPEC) shows the refinement of the behavior $spec' = \texttt{requires}\ p'\ \texttt{ensures}\ q'$ by the behavior $spec = \texttt{requires}\ p\ \texttt{ensures}\ q$. For the behavior *spec* to refine $spec'$, its precondition p must imply the precondition p', i.e. $p \Rightarrow p'$, and the opposite must be true for their postconditions, i.e. $q' \Rightarrow q$. That is the subevent can *strengthen* the precondition of its superevent and *weaken* its postcondition which is the inverse of classical refinement in class subtyping [23] where a subclass weakens the precondition of its superclass and strengthens its postcondition. Such inverse relation of behaviors is necessary in *Ptolemy*$_{\mathbb{S}}$ to allow an observer of a superevent to run upon announcement of its subevents. Also unlike *Ptolemy*$_{\mathbb{S}}$'s refining, the classical refining is for blackbox contracts and does not directly apply to greybox translucid contracts [22] and especially their assumes block [29] with control effect specifications.

The assumes block *se* of the translucid contract of an event refines the assumes block se' of the contract of its superevent, i.e. $se' \trianglelefteq se$, if: *(a)* each specification expression in *se* refines its *corresponding* specification expression in se' and *(b)* each program expression in *se* refines its corresponding program expression in *se*. The rule (R-SPEC) for refinement of behaviors also applies for refinement of specification expressions since they similarly are behavior specifications with a precondition and postcondition [29]. A specification expression in a subevent can strengthen the precondition of its corresponding specification expression in its superevent and weaken its postcondition. For a program expression to refine another program expression, they must textually match. The rule (R-VAR) checks for textual matching of variable names using the auxiliary function *textualMatch*. For other program expressions, such as invoke and conditional, their refinement boils down to the refinement of their subexpressions, as in rules (R-INVOKE), (R-DEFINE) and (R-IF).

To illustrate, the translucid contract of `AndEv`, on lines 20–26 in Figure 4, refines the contract of `ExpEv`, on lines 2–8. This is because *(i)* the precondition $left != \texttt{null}\ \&\&\ right != \texttt{null}\ \&\&\ node != \texttt{null}$ of `AndEv` implies the precondition $node != \texttt{null}$ of `ExpEv` and the postcondition $node.equals(\texttt{old}\ (node))$ of `ExpEv` implies the same postcondition of `AndEv`, and thus using the rule (R-SPEC) the behavior of `AndEv` refines the behavior of `ExpEv`; *(ii)* the program expression $\texttt{next}.\texttt{invoke}()$ of `AndEv`, on line 22, refines its corresponding program expression of `ExpEv`, on line 4, using (R-INVOKE) and (R-VAR) and specification expression $\texttt{requires}\ \texttt{next}.node().left == \texttt{old}\ (\texttt{next}.node().left) \&\& \texttt{next}.node().right == \texttt{old}\ (\texttt{next}.node().right)\ \texttt{ensures}\ \texttt{next}.node().parent == \texttt{old}\ (\texttt{next}.node().parent)$ of `AndEv`, on lines 23–24, refines its corresponding specification expression $\texttt{requires}\ \texttt{true}\ \texttt{ensures}\ \texttt{next}.node().parent == \texttt{old}\ (\texttt{next}.node().parent)$ in `ExpEv`, on lines 5–6, using (R-SPEC).

However, the translucid contract of `AndEv` does not refine the contract of `BinEv`, on lines 11–17, because the postcondition **true** of `BinEv` does not imply the postcondition of `AndEv`. Changing the postcondition of `BinEv` to $\texttt{next}.node().parent == \texttt{old}\ (\texttt{next}.node().parent)$ makes the contract of `BinEv` refine the contract of `ExpEv`.

Textual matching of program expressions is a simpler alternative to complex higher order logic or trace verification techniques with its tradeoffs [29]. Textual matching works because *Ptolemy*$_{\mathbb{S}}$'s semantics enforces depth subtyping, ensuring that a redeclaring context variable in an event is a subtype of the redeclared context in its superevents and a **next** variable in the contract of an event is a subtype of the next variable in the contract of its superevent.

The refining relation $\trianglelefteq$ defines the refinement for corresponding program and specification expressions, that is only *structurally similar* contracts may refine each other. Two translucid contracts are structurally similar if for each specification (program) expression in the assumes block of one, a possibly different specification (program) expression exists in the assumes block of the other at the same location. *Ptolemy*$_{\mathbb{S}}$'s structural similarity for the refining relation allows definition of *Ptolemy*$_{\mathbb{S}}$'s event specification inheritance, in [26], such that it statically guarantees the refining relation by combining translucid contracts of an event and its superevents in a subtyping hierarchy.

3.3 Non-Decreasing Relation of Observers' Execution

Ptolemy$_{\mathbb{S}}$ limits the arbitrary execution order of observers of an event and its superevents by enforcing a non-decreasing relation on execution orders of observers. In the non-decreasing order , an observer of an event runs before an observer of its superevent. *Ptolemy*$_{\mathbb{S}}$'s semantics for **announce**, **invoke**, **register** and **unregister** expressions and the relation of return types of events in an event hierarchy guarantee the non-decreasing order.

In *Ptolemy*$_{\mathbb{S}}$, a subject announces an event *ev* using the announce expression $\texttt{announce}\ ev(e*)\{e'\}$. The announce expression evaluates parameters $e*$ to values $v*$, creates an event closure for the event *ev* and binds values $v*$ to context variables of *ev* in the closure. The announce expression also creates, in the event closure, a chain containing registered observers of *ev* and observers of *all its superevents* and runs the first observer in the chain. To construct the chain, the announce expression adds observers of the event *ev* to an empty chain followed by adding observers of the direct superevent of *ev* and recursively continues until it reaches the root event **Event**[3]. The event body e' is added to the end of the chain.

By construction, the announce expression ensures that an observer of an event shows up before an observer of its superevent in the chain which basically is the non-decreasing order of observers.

[3] **Event** is not accessible to programmers and does not have observers, as a simple design choice, to not allow programmers to affect behaviors of events of a system by defining a specification for **Event**.

Observers of the same event in the chain, *maintain* among themselves, the same order as their dynamic registration order, i.e. an observer registered earlier shows up in the chain before the ones registered later. This makes *Ptolemy*$_{\mathbb{S}}$ backward compatible with its earlier versions [5, 6, 13] that do not support event subtyping. The expression **next** is a placeholder for an event closure and the type **thunk** *ev* is the type of the event closure of an event *ev*.

After construction of the chain and running the first observer in the chain, by the announce expression, observers in the chain can invoke each other using an invoke expression e.**invoke**(). The invoke expression evaluates e to an event closure containing the chain of observers and runs the next observer in the chain, which is according to the non-decreasing order. For observers to run in the non-decreasing order, the return type of an observer of an event must be a supertype of the return type of the observers of its superevent. *Ptolemy*$_{\mathbb{S}}$'s static semantics, in Section B, guarantees this by ensuring that the return type of an event is a supertype of the return type of its superevent.

Upon announcement of an event, only registered observers of the event and its superevents run. In *Ptolemy*$_{\mathbb{S}}$, observers show interest in events through binding declarations and register to handle the events. A binding declaration **when** *ev* **do** *m* in an observer says to run the observer handler method *m* when an event of type *ev* is announced. The expression **register**(e) evaluates e to an object and adds it to the list of observers $A[ev]$ for each event type *ev* that is named in binding declarations of the observer, and **unregister**(e) removes the object e from the list of observers of those events. The announce expression for an event *ev* recursively concatenates the list of observers $A[ev]$ of the event *ev* and the list of observers of its superevents to construct the chain of observers.

3.4 Refining + Non-decreasing

Any of refining or non-decreasing relations alone cannot solve both combinatorial reasoning and behavior invariance problems. With the refining alone, because of the arbitrary execution order of observers, still up to $n!$ possible execution orders of n observers of the event and observers of its superevents should be considered in reasoning, which threatens its tractability; changes in execution orders of observers of the event or observers of its superevents can still invalidate any previous reasoning, which threatens modularity of reasoning; and observers of events in a subtyping hierarchy still could be forced to satisfy the same behavior, which threatens reuse. A trivial refining relation in which events of a hierarchy satisfy the same behavior enables modular reasoning, however it is undesirable as it prevents reuse of event types, their specifications [28] and observers [14, 15].

With the non-decreasing alone, because of unrelated behaviors of observers, observers of events in a subtyping hierarchy may still be forced to satisfy the same behavior and any changes in behaviors of superevents of an event could invalidate any previous reasoning about subjects and observers of the event.

Interestingly, reversing both refining and non-decreasing relations still allows modular reasoning. To reverse these relations, the translucid contract of a superevent refines the contract of its subevent and an observer of a superevent runs before any observer of its subevent. We chose the current design, as it seemed more natural, to us, for observers of an already announced event to run before observers of its superevents.

4. Modular Reasoning

This section formalizes *Ptolemy*$_{\mathbb{S}}$'s Hoare logic for modular reasoning, its conformance relation for subjects and observers and soundness of its reasoning technique.

Ptolemy$_{\mathbb{S}}$'s refining and non-decreasing relations enable its modular reasoning about subjects and observers of an event, as shown in Figure 8. The main idea is to use the translucid contract of an event as a sound approximation of the behaviors of its observers and observers of its superevents to reason about:

(1) a subject of the event, especially its **announce** expression, independent of its observers and observers of its superevents and their execution orders; and

(2) an observer of the event, especially its **invoke** expressions, independent of its subjects as well as observers it may invoke and their execution orders.

reasoning judgement: $\boxed{\Gamma \vdash \{p\}\ e\ \{q\}}$

(V-ANNOUNCE)
$$\frac{\begin{array}{c}(c\ \mathbf{event}\ ev\ \mathbf{extends}\ ev'\{(t\ var)^*\ contract\}) \in CT\\ contract = \mathbf{requires}\ p\ \mathbf{assumes}\ \{se\}\ \mathbf{ensures}\ q\\ topContract(ev) = \mathbf{requires}\ p'\ \mathbf{assumes}\ \{se'\}\ \mathbf{ensures}\ q'\\ \Gamma \vdash \{p'[e^*/var^*]\}\ e'\ \{q'[e^*/var^*]\}\end{array}}{\Gamma \vdash \{p[e^*/var^*]\}\ \mathbf{announce}\,ev(e^*)\,\{e'\}\ \{q[e^*/var^*]\}}$$

(V-INVOKE)
$$\frac{\begin{array}{c}\mathbf{thunk}\ ev = \Gamma(\mathbf{next})\\ (c\ \mathbf{event}\ ev\ \mathbf{extends}\ ev'\{form^*\ contract\}) \in CT\\ contract = \mathbf{requires}\ p\ \mathbf{assumes}\ \{se\}\ \mathbf{ensures}\ q\end{array}}{\Gamma \vdash \{p\}\ \mathbf{next}.\mathbf{invoke}()\ \{q\}}$$

(V-REFINING)
$$\frac{\Gamma \vdash \{p\}\ e\ \{q\}}{\Gamma \vdash \{p\}\ (\mathbf{refining\ requires}\ p\ \mathbf{ensures}\ q\ \{\ e\ \})\ \{q\}}$$

(V-SPEC)
$$\Gamma \vdash \{p\}\ \mathbf{requires}\ p\ \mathbf{ensures}\ q\ \{q\}$$

(V-CONSEQ)
$$\frac{p \Rightarrow p' \qquad q' \Rightarrow q \qquad \{p'\}\ e\ \{q'\}}{\Gamma \vdash \{p\}\ e\ \{q\}}$$

Figure 8. Select reasoning rules in *Ptolemy*$_{\mathbb{S}}$'s Hoare [24] logic, inspired by [7, 29].

Figure 8 shows *Ptolemy*$_{\mathbb{S}}$'s Hoare logic [24] for modular reasoning about behaviors of subjects and observers. *Ptolemy*$_{\mathbb{S}}$'s reasoning rules use a reasoning judgement of the form $\Gamma \vdash \{p\}\ e\ \{q\}$ that says the Hoare triple $\{p\}\ e\ \{q\}$ is provable using the variable typing environment Γ, which maps variables to their types. The judgement $\Gamma \vdash \{p\}\ e\ \{q\}$ is valid, written as $\Gamma \models \{p\}\ e\ \{q\}$, if for every state σ that agrees with type environment Γ, if p is true in σ, i.e. $\sigma \models p$, and if the execution of e terminates in a state σ', then $\sigma' \models q$. This definition of validity is for partial correctness where termination is not guaranteed. *Ptolemy*$_{\mathbb{S}}$'s reasoning rules use a fixed class table *CT*, which is a set of the program's class and event type declarations. The notation $ep[e^*/var^*]$ denotes replacing variables var^* with e^* in the expression ep. *Ptolemy*$_{\mathbb{S}}$'s rules for reasoning about standard object-oriented expressions remain the same as in previous work [24, 29–31] and are omitted.

In Figure 8, the rule (V-ANNOUNCE) reasons about the behavior of an announce expression, in a subject. The rule says that the behavior of an announce expression announcing an event *ev* is the behavior **requires** *p* **ensures** *q* of the translucid *contract* of the event *ev*. To use the precondition *p* of the contract and its postcondition *q* in the scope of the announce expression, their context variables var^* are replaced by arguments e^* of the announce expression [27]. The rule (V-ANNOUNCE) does not require and is independent of any knowledge of individual observers of *ev* or observers of its superevents, their implementations or execution orders which in turn makes it modular and tractable.

To illustrate (V-ANNOUNCE), reconsider verification of the assertion Φ for the announce expression of `AndEv`, on lines 4–7 of Figure 1. Using the translucid contract of `AndEv`, on lines 20–26, the conclusion of (V-ANNOUNCE), replaces parameters `e`, `e.left` and `e.right` of the announce expression for context variables of `node`, `left` and `right` of `AndEv` in the precondition and postcondition of the contract of `AndEv` and yields the Hoare triple:

$$\begin{array}{l}\Gamma \vdash \{e.left != \texttt{null} \ \&\& \ e.right != \texttt{null} \ \&\& \ e != \texttt{null}\} \\ \quad\quad \textbf{announce}\ AndEv(e,\ e.left,\ e.right) \\ \quad\quad \{e.left.accept(\textbf{this});\ e.right.accept(\textbf{this});\} \\ \quad \{e.equals(\textbf{old}\ (e))\}\end{array}$$

The above judgement says, if `e`, `e.left` and `e.right` are not null, the expression e and its state remain the same after announcement and handling of `AndEv`, i.e. $e.equals(\textbf{old}\ (e))$, which is exactly the assertion Φ we wanted to verify.

The rule (V-INVOKE) reasons about the behavior of an invoke expression, in an observer. The rule says that the behavior of an invoke expression in an observer of the event *ev*, is the behavior of the translucid *contract* of *ev*. The type of the event that the observer handles, i.e. *ev*, is part of the type of the event closure **next**. The function $\Gamma(\textbf{next})$ returns the type of the next expression in the typing environment Γ. Recall that the event closure next is passed as a parameter to each observer handler method. Again, the rule (V-INVOKE) does not require and is independent on any knowledge about subjects of the event *ev* or observers it may invoke in the chain of observer **next** and thus is modular and tractable.

The rule (V-REFINING) says that the behavior of the body e of a refining expression is the behavior of its specification expression **requires** p **ensures** q. This is true, because the body of the refining expression claims to refine its specification. The rule (V-SPEC) is straightforward [29] and the rule (V-CONSEQ) is standard [24].

4.1 Soundness of Reasoning

In *Ptolemy*$_{\mathbb{S}}$'s the translucid contract of an event is a sound approximation of behaviors of its subjects and observers independent of observers of the event, observers of its superevents and their execution orders. This is sound because of the following:

1. Conformance of each observer and subject of an event to the translucid contract of the event;
2. Refining relation among specifications of the event and its superevents; and
3. Non-decreasing relation on execution orders of observers of the event and observers of its superevents.

For a greybox translucid contract of an event, *all* subjects and observers of the event must conform to the contract of the event. This is different from a blackbox method specification, e.g. in JML, in which only a single method has to respect a contract [6, 23]. *Ptolemy*$_{\mathbb{S}}$'s semantics, in Appendices A and B, guarantees the conformance, using a combination of type checking and runtime assertion checking. *Ptolemy*$_{\mathbb{S}}$'s event specification inheritance [26], statically guarantees the refining relation and *Ptolemy*$_{\mathbb{S}}$'s dynamic semantics guarantees the non-decreasing relation. Figure 5 shows the interplay of conformance, refining and non-decreasing relations.

4.1.1 Conforming Observers

DEFINITION 4.1. *(Conforming observer) For an event type ev with a translucid contract* $\mathscr{G} = (\textbf{requires}\ p\ \textbf{assumes}\ \{se\}\ \textbf{ensures}\ q)$, *its observer handler method m with its implementation e is conforming if and only if there exists a typing environment* Γ *such that:*

(i). $\Gamma \models \{p\}\ e\ \{q\}$
(ii). $se \sqsubseteq_s e$

where Figure 9 defines the structural refinement relation $\sqsubseteq_s$ *between the assumes block se and the body e of its observer.*

Structural refinement relation: $\boxed{\Gamma \vdash se \sqsubseteq_s e}$

(S-REFINING)
$$\Gamma \vdash spec \sqsubseteq_s \textbf{refining}\ spec\ \{e\}$$

(S-INVOKE)
$$\frac{\Gamma \vdash se \sqsubseteq_s e}{\Gamma \vdash se.\textbf{invoke}() \sqsubseteq_s e.\textbf{invoke}()}$$

(S-VAR)
$$\frac{textualMatch(var', var)}{\Gamma \vdash var' \sqsubseteq_s var}$$

(S-ANNOUNCE)
$$\frac{\Gamma \vdash se* \sqsubseteq_s e* \qquad \Gamma \vdash se \sqsubseteq_s e}{\Gamma \vdash \textbf{announce}\ ev(se*)\{se\} \sqsubseteq_s \textbf{announce}\ ev(e*)\{e\}}$$

(S-EITHEROR)
$$\frac{\Gamma \vdash se_1 \sqsubseteq_s e \vee \Gamma \vdash se_2 \sqsubseteq_s e}{\Gamma \vdash \textbf{either}\ \{se_1\}\ \textbf{or}\ \{se_2\} \sqsubseteq_s e}$$

(S-DEFINE)
$$\frac{\Gamma \vdash se_1 \sqsubseteq_s e_1 \qquad \Gamma, var : t \vdash se_2 \sqsubseteq_s e_2}{\Gamma \vdash t\ var = se_1; se_2 \sqsubseteq_s t\ var = e_1; e_2}$$

(S-IF)
$$\frac{\Gamma \vdash sp \sqsubseteq_s ep \qquad \Gamma \vdash se_1 \sqsubseteq_s e_1 \qquad \Gamma \vdash se_2 \sqsubseteq_s e_2}{\Gamma \vdash \textbf{if}(sp)\{se_1\}\ \textbf{else}\{se_2\} \sqsubseteq_s \textbf{if}(ep)\{e_1\}\ \textbf{else}\{e_2\}}$$

Figure 9. Select rules for structural refinement $\sqsubseteq_s$ [5, 29].

Definition 4.1 says that for an observer handler method of an event *ev* to be conforming, its implementation e must satisfy the precondition p and postcondition q of the translucid contract of the event, i.e. requirement *(i)*. An expression e satisfies a precondition p and a postcondition q in a typing environment Γ, written as $\Gamma \models \{p\}\ e\ \{q\}$, if and only if for every program state σ that agrees with the type environment Γ, if the precondition p is true in σ, and if the execution of e terminates in a state σ', then q is true in σ'. Currently *Ptolemy*$_{\mathbb{S}}$ uses runtime assertions to check for satisfaction of preconditions and postconditions of a contract by its observers. Static verification techniques could also be used to check for such satisfaction. Figure 10 shows the conforming observer `Evaluator` and its observer handler method `evalAndExp`, on lines 21–32. In `evalAndExp`, assertions on lines 22 and 31 check for preconditions and postconditions of the contract of `AndEv` on lines 2 and 8.

Definition 4.1, also requires the implementation e of a conforming observer to structurally refine the assumes block *se* of its translucid contract, i.e. requirement *(ii)*. The structural refinement $\sqsubseteq_s$ guarantees that an observer of an event, in its implementation, has the control effects exposed in its translucid contract [5, 6] using its program expressions. Figure 9 shows select rules for *Ptolemy*$_{\mathbb{S}}$'s structural refinement.

The implementation e of an observer handler method structurally refines the assumes block *se* of its translucid contract if: *(a)* for each specification expression *spec* in *se* there is a corresponding **refining** expression in e with the same specification and *(b)* for each program expression in *se*, there is a corresponding textually matching program expression in e. The rule (S-REFINING) checks for structural refinement of a specification expression by a refining expression. (S-VAR) checks for textual matching of variable names using the auxiliary function *textualMatch*. For other program expressions, structural refinement boils down to structural refinement of their subexpressions. The rule (S-EITHEROR) allows an observer to choose between behaviors in its either-branch or its or-branch. Similar to the refining relation, structural refinement requires structural similarity between the implementation of a conforming observer and the assumes block of its contract.

In Figure 10, the assumes block on lines 4–6 is structurally refined by the implementation of the conforming observer `evalAndExp`, on lines 23–32, because the program expression **next**.**invoke**() on line 4 is structurally refined by the program expression in the implementation on line 23 and the specification expression on lines 5–6 is refined by a refining expression with the same specification on lines 25–29. Structural refinement guarantees that the implementation of `evalAndExp` has, in its implementation, a **next**.**invoke**() expression as its control effect, as specified by the program expression **next**.**invoke**() in its translucid contract.

A refining expression claims that its body satisfies its specification. *Ptolemy*$_{\mathbb{S}}$ uses runtime assertions to check this claim. In Figure 10, runtime checks on lines 24 and 30 check that the body of the refining expression satisfies its precondition and postcondition on lines 26 and 27.

Though similar, in the structural refinement $\sqsubseteq_s$ the implementation of an observer refines the assumes block of the translucid contract of its event, whereas in the refining relation $\trianglelefteq$ the contract of an event refines the contract of its superevent. A specification expression in a contract is structurally refined by a refining expression in $\sqsubseteq_s$ whereas it is refined by another specification expression in $\trianglelefteq$.

4.1.2 Conforming Subjects

DEFINITION 4.2. *(Conforming subject) For an event type ev with a translucid contract* $\mathcal{G} = (\texttt{requires}\ p\ \texttt{assumes}\ \{se\}\ \texttt{ensures}\ q)$, *its subject with an announce expression* $\texttt{announce}\ ev(e*)\{e'\}$ *in its implementation, is conforming if and only if:*
$\Gamma \models \{p'\}\ e'\ \{q'\}$ *where* $\texttt{requires}\ p'\ \texttt{assumes}\ \{se'\}\ \texttt{ensures}\ q' = topContract(ev)$

The definition says that for a subject of *ev* to be conforming its event body e' must satisfy the precondition p' and postcondition q' of the translucid contract of the event on top of the subtyping hierarchy of *ev*, right before the root event **Event**. The auxiliary function *topContract* returns the translucid contract of this event. As shown in Figure 5, this is necessary for the non-decreasing relation in which observers of the event and observers of its superevent run before the event body e' in the chain of observers. Figure 10 shows the conforming subject `ASTVisitor`, on lines 10–19. Runtime assertions on lines 13 and 16 check for satisfaction of the precondition and postcondition of the top contract of `AndEv`, i.e. the translucid contract of `ExpEv`, by the event body.

4.1.3 Soundness Theorem

Theorem 4.3 formalizes soundness of *Ptolemy*$_{\mathbb{S}}$'s Hoare logic.

THEOREM 4.3. *(Soundness of Ptolemy$_{\mathbb{S}}$'s Hoare logic) Ptolemy$_{\mathbb{S}}$'s Hoare logic, in Figure 8, is sound for conforming Ptolemy$_{\mathbb{S}}$ programs. In other words, any Hoare triple provable using Ptolemy$_{\mathbb{S}}$'s logic, i.e.* $\Gamma \vdash \{p\}\ e\ \{q\}$, *is a valid triple, i.e.* $\Gamma \models \{p\}\ e\ \{q\}$.

The proof is based on induction on the number of events in a subtyping hierarchy and the number of their observers and uses conformance, refining and non-decreasing relations. Full proof of the theorem can be found in Section D.

```
1  void event AndEv extends BinEv { ..
2   requires left != null && right != null && node != null
3   assumes {
4    next.invoke();
5    requires next.node().left!=null&&next.node().right!=null
6    ensures next.node().parent == old(next.node().parent);
7   }
8   ensures node.equals(old(node))
9  }
10 class ASTVisitor {
11  void visit(AndExp e) {
12   announce AndEv(e, e.left, e.right) {
13    assert(e != null);
14    e.left.accept(this);
15    e.right.accept(this);
16    assert(node.equals(old(node)));
17   }
18  } ..
19 }
20 class Evaluator { ..
21  void evalAndExp (AndEv next) {
22   assert(next.node().left!=null&&next.node().right!=null
        &&next.node()!=null);
23   next.invoke();
24   assert(next.node().left!=null&&next.node().right!=null);
25   refining
26   requires next.node().left!=null&&next.node().right!=null
27   ensures next.node().parent == old(next.node().parent){
28    BoolVal b1 = (BoolVal) valStack.pop();
29   }
30   assert(next.node().parent == old(next.node().parent));
31   assert(next.node().equals(old(next.node())));
32  }
33  when AndEv do evalAndExp; ..
34 }
```

Figure 10. Conforming `Evaluator` and `ASTVisitor`.

4.2 Revisiting Reasoning about Announce and Invoke

Ptolemy$_{\mathbb{S}}$'s reasoning rules (V-ANNOUNCE) and (V-INVOKE) are sound because the conformance, refining and non-decreasing relations allow, in any chain of observers, the implementation of an invoked observer to be inlined in place of invoke expressions of its invoking observer *without violating* the precondition and postcondition of the invoking observer. This in turn allows the chain of observers of an event and observers of its superevents, starting from the event body at the end of the chain back to its beginning, to be recursively inlined in an announce expression without violating the precondition and postcondition of the contract of the event.

To illustrate, reconsider reasoning about the behavior of $\texttt{announce}\ AndEv(e,\ e.left,\ e.right)$, in Figure 1. Upon announcement of `AndEv`, if there are no observers of `AndEv` or observers of its superevents `BinEv` or `ExpEv` in the chain of observers, then the event body $e.left.accept(\texttt{this});\ e.right.accept(\texttt{this})$ executes. The subject `ASTVisitor` of `AndEv` is conforming and thus the event body satisfies the behavior of the contract of `ExpEv`, which is the top event in the hierarchy of `AndEv`. That is, the event body satisfies the precondition $node\ != \texttt{null}$ and the postcondition $node.equals(\texttt{old}\ (node))$ of `ExpEv` after the context *node* is replaced with parameter *e* of the announce expression:

$$\begin{array}{l}\text{(H-BODY)}\\ \Gamma \models \{e\ != \texttt{null}\}\\ \qquad e.left.accept(\texttt{this});\ e.right.accept(\texttt{this});\\ \quad \{e.equals(\texttt{old}\ (e))\}\end{array}$$

The refining relation guarantees that the behavior of `AndEv` refines the behavior of `ExpEv`. That is the precondition of `AndEv` implies the precondition of `ExpEv`, i.e. $left! = \texttt{null}\ \&\&\ right! = \texttt{null}\ \&\&\ node! = \texttt{null} \Rightarrow node\ ! = \texttt{null}$, and the opposite is true for their postconditions, i.e. $node.equals(\texttt{old}\ (node)) \Rightarrow node.equals(\texttt{old}\ (node))$. Using these implications, and the rule (V-CONSEQ) and after replacing the context *node* with *e*, one can conclude that the event body satisfies the behavior of `AndEv`:

$$\begin{array}{l}\Gamma \models \{e.left! = \texttt{null}\ \&\&\ e.right! = \texttt{null}\ \&\&\ e! = \texttt{null}\}\\ \qquad e.left.accept(\texttt{this});\ e.right.accept(\texttt{this});\\ \quad \{e.equals(\texttt{old}\ (e))\}\end{array}$$

Since the event body is the only observer that executes upon announcement of `AndEv`, the announce expression can be replaced with the event body:

(H-ANNOUNCE-BODY)

$$
\begin{array}{l}
\Gamma \models \{e.left! = \mathtt{null}\ \&\&\ e.right! = \mathtt{null}\ \&\&\ e! = \mathtt{null}\} \\
\qquad \mathtt{announce}\ AndEv(e,\ e.left,\ e.right) \\
\qquad \{e.left.accept(\mathtt{this});\ e.right.accept(\mathtt{this});\} \\
\quad \{e.equals(\mathtt{old}\ (e))\}
\end{array}
$$

The judgement (H-ANNOUNCE-BODY) says the announce expression of `AndEv` with event body as its only observer satisfies the behavior of the translucid contract of `AndEv`.

However, the event body may not be the only observer of `AndEv`. Consider observers `evaluator` and `tracer` of event `AndEv` and `ExpEv` and the event body of `AndEv`, shown as $\mathscr{B}$`(AndEv)`, run in a chain χ_1 : `evaluator` $\rightharpoonup$ `tracer` $\rightharpoonup$ $\mathscr{B}$`(AndEv)`. Again, conformance of `ASTVisitor` means that the event body satisfies the behavior of the contract of `ExpEv`, i.e. (H-BODY). Recall that an observer of an event and the invoke expressions in its implementation have the precondition and postcondition of the contract of the event. The precondition of the invoke expression in the implementation of `tracer` implies the precondition of the event body, i.e. $node! = \mathtt{null} \Rightarrow node! = \mathtt{null}$ and the postcondition of the event body implies the postcondition of the invoke expression, i.e. $node.equals(\mathtt{old}\ (node)) \Rightarrow node.equals(\mathtt{old}\ (node))$. This in turn allows the event body, in grey, to be inlined in the place of the invoke expression in the implementation of `tracer`, in Figure 3, without violating the precondition and postcondition of `tracer`:

(H-TRACER)

$$
\begin{array}{l}
\Gamma \models \{e\ ! = \mathtt{null}\} \\
\qquad e.left.accept(\mathtt{this});\ e.right.accept(\mathtt{this}); \\
\qquad \mathtt{refining\ requires\ true} \\
\qquad \mathtt{ensures}\ e.parent == \mathtt{old}\ (e.parent)\{..\} \\
\quad \{e.equals(\mathtt{old}\ (e))\}
\end{array}
$$

Using the refining relation, the precondition of `AndEv` implies the precondition of `ExpEv` and the opposite is true for their postconditions. This means the precondition of the invoke expression in the implementation of `evaluator` implies the precondition of `tracer`, i.e. $left! = \mathtt{null}\ \&\&\ right! = \mathtt{null}\ \&\&\ node! = \mathtt{null} \Rightarrow node\ ! = \mathtt{null}$, and the postcondition of `tracer` implies the postcondition of the invoke expression in `evaluator`, i.e. $node.equals(\mathtt{old}\ (node)) \Rightarrow node.equals(\mathtt{old}\ (node))$. This allows the implementation of `tracer` in (H-TRACER) to be inlined, in grey, in place of the invoke expression in `evaluator` without violating its precondition and postcondition of `evaluator`:

(H-EVALUATOR)

$$
\begin{array}{l}
\Gamma \models \{e.left! = \mathtt{null}\ \&\&\ e.right! = \mathtt{null}\ \&\&\ e! = \mathtt{null}\} \\
\qquad e.left.accept(\mathtt{this});\ e.right.accept(\mathtt{this}); \\
\qquad \mathtt{refining\ requires\ true} \\
\qquad \mathtt{ensures}\ e.parent == \mathtt{old}\ (e.parent)\{..\}; \\
\qquad \mathtt{refining} \\
\qquad \mathtt{requires}\ e.left! = \mathtt{null}\ \&\&\ e.right! = \mathtt{null} \\
\qquad \mathtt{ensures}\ e.parent == \mathtt{old}\ (e.parent)\{..\}; \\
\quad \{e.equals(\mathtt{old}\ (e))\}
\end{array}
$$

Since the announcement of `AndEv` causes the chain χ_1 to run, the inlined chain of observers in (H-EVALUATOR) can be replaced with the announce expression:

(H-ANNOUNCE-χ_1)

$$
\begin{array}{l}
\Gamma \models \{e.left! = \mathtt{null}\ \&\&\ e.right! = \mathtt{null}\ \&\&\ e! = \mathtt{null}\} \\
\qquad \mathtt{announce}\ AndEv(e,\ e.left,\ e.right) \\
\qquad \{e.left.accept(\mathtt{this});\ e.right.accept(\mathtt{this});\} \\
\quad \{e.equals(\mathtt{old}\ (e))\}
\end{array}
$$

The judgement (H-ANNOUNCE-χ_1) says that the behavior of the announce expression of `AndEv` with the chain of observers χ_1 satisfies the behavior of the contract of `AndEv`.

(H-ANNOUNCE-BODY) and (H-ANNOUNCE-χ_1) say the behavior a chain of observers of `AndEv` and observers of its superevents, can be approximated with the precondition and postcondition of the translucid contract of the `AndEv` which is what the rule (V-ANNOUNCE) in *Ptolemy*$_{\mathbb{S}}$'s reasoning logic says. A similar justification is true for the rule (V-INVOKE).

5. Applicability

Our proposed modular reasoning technique is not exclusive to *Ptolemy*$_{\mathbb{S}}$ and could be adapted to similar AspectJ-like [1] event-based systems such as join point types (JPT) [15] and join point interfaces (JPI) [14]. Application of our reasoning technique to join point interfaces could be found in Section C.

With join point types, a subject (base) exhibits a join point type (event) using an **exhibits** statement and aspects (observers) advise the event and handle it using **advises** statements. A join point type can extend another join point type, inherit its context variables, and add to them through width subtyping. Exhibiting a join point type causes its aspects and aspects of its super join point types to run in a chain where aspects can invoke each other, using **proceed** statements. The execution order of aspects is specified using precedence declarations. Join point types do not support depth subtyping, however, this does not affect the applicability of *Ptolemy*$_{\mathbb{S}}$'s reasoning technique to them.

```
1  joinpointtype AndEv extends BinEv {
2  /*@ requires node!=null && left!=null &&right!=null;
3    @ model_program {
4    @  proceed(next);
5    @  requires node.left!=null && node.right!=null;
6    @  ensures node.parent == old(node.parent);
7    @ }
8    @ ensures node.equals(old(node)); */
9  }
10 class ASTVisitor exhibits AndEv,.. {
11  void visit(AndExp e) {
12   exhibits new AndEv(e, e.left, e.right) {
13    e.left.accept(this);
14    e.right.accept(this);
15   }; ..
16  } ..
17 }
18 aspect Evaluator advises AndEv,.. { ..
19  void around(AndEv jp) {
20   proceed(jp);
21   refining
22    requires node.left!=null && node.right!=null;
23    ensures node.parent == old(node.parent){
24    .. //same as before
25   }
26  } ..
27 }
```

Figure 11. Join point type `AndEv` and its translucid contract

Figure 11 shows parts of the expression language example rewritten using join point types where the subject `ASTVisitor` exhibits a join point instance `AndEv`, on lines 12–15, and the observer `Evaluator` advises the join point, on lines 19–26. `Evaluator` invokes the next observer in the chain of observers using a proceed statement on line 20, which takes as argument a join point instance `jp` of join point type `AndEv`. The join point type `AndEv` is declared on lines 1–9 and extends the join point type `BinEv`.

Figure 11 shows the syntactic adaptation of the translucid contract of the join point type `AndEv`, on lines 2–8, using a JML-like syntax. JML syntax is specifically chosen to minimize required syntactic changes. In a contract of a join point type, a JML model program [29] is similar to an assumes block and a proceed statement is equivalent to an invoke expression [5]. A variable **next** in the contract of a join point type is a placeholder for join point instances of that type, which contains values of its contexts.

Although, a translucid contract of a join point type uses JML's syntax, its verification is completely different from JML. This is because a JML contract specifies the behavior and structure of only a single method whereas a translucid contract of a join point type specifies all bases and aspects of the join point type. Consequently, for the conformance relation, for each join point type, all of its bases and aspects must conform to the translucid contract of their join point type, i.e. structurally refine the contract and satisfy its preconditions and postconditions. Type checking rules of join point types could be augmented to check for structural refinement and runtime assertions could be added to bases and aspects to check for their satisfaction of preconditions and postconditions of their contract and their specification expressions. In addition to syntactic adaptations of structural refinement, the rule (S-VAR) should be slightly modified to allow for structural refinement of placeholder variables **next** by join point instance variables. Unlike *Ptolemy*$_{\mathbb{S}}$ in which a variable next is structurally refined by a textually matching variable next, in join point types a variable next in a contract of a join point type is structurally refined by a join point instance variable in the implementation of an observer if their types are the same. For example, in Figure 11, the variable next in the translucid contract of `AndEv`, on line 4, is structurally refined by the join point instance variable `jp` in the observer `Evaluator`, on line 20, because they both are of the same type `AndEv`.

Another difference between translucid contracts and JML contracts is that, JML requires model programs of a type and its supertype to be the same [29], whereas in translucid contracts the assumes block of an event refines the assumes block of its superevent. Consequently, for the refining relation, *Ptolemy*$_{\mathbb{S}}$'s specification inheritance [26] could be adapted to join point types, mostly through syntactic adaptations, to statically guarantee the refining relation between translucid contracts of a join point type and its super type.

For the non-decreasing relation, precedence declarations of aspects could be statically checked to ensure that an aspect of a join point type runs before aspects of its super join point type or execution of aspects can be reordered dynamically at runtime to guarantee the non-decreasing relation.

A similar technique, with several adaptations, could be applied to join point interfaces, due to similarities of event announcement, handling and subtyping models of join point types and join point interfaces [26].

6. Modular Reasoning about Control Effects

Ptolemy$_{\mathbb{S}}$ not only enables modular reasoning about behaviors of observers of an event but also their control effects [5, 20] in the presence of event subtyping. In *Ptolemy*$_{\mathbb{S}}$, similar to Aspect-like [1] languages, observers run in a chain and invoke each other using an **invoke** expression. This in turn means an observer of an event can skip the execution of other observers of the event or observers of its superevents, including the event body, by not executing its invoke expression. Understanding the invocations among observers of an event and its superevents in a chain of observers, falls under the category of modular reasoning about control effects of observers.

As an example of modular reasoning about control effects of observers consider static verification of the control effect assertion Ψ that says: *upon announcement and handling of* `AndEv`*, its event body, on lines 5–6 of Figure 1 will be executed and will not be skipped*[4]. This is important because if the execution of the event body of `AndEv` is skipped, the right and left children of an `AndExp` expression and subtrees recursively rooted in these children are not going to be visited. The execution of the body of `AndEv`, shown as $\mathscr{B}$`(AndEv)`, could be skipped in a chain of observers if any of observers of `AndEv` or observers of its superevents `BinEv` or `ExpEv`, which run before the event body, skip the execution of their invoke expression and break the invocation chain. For example, in chain χ_2: `evaluator` $\rightharpoonup$ `tracer` $\rightharpoonup$ $\mathscr{B}$`(AndEv)`, the execution of $\mathscr{B}$`(AndEv)` is skipped if any or both invoke expressions in the implementations of `evaluator`, on line 59 of Figure 3, or `tracer`, on line 43, goes missing.

To reason about control effects of announcement of an event, the control effects of all of its observers and observers of its superevents for their various execution orders must be understood, especially regarding the execution of their invoke expressions. Such reasoning is dependent on control effects of individual observers of the event and observers of its superevents and any changes in these control effects can invalidate any previous reasoning, which threatens its modularity.

Ptolemy$_{\mathbb{S}}$'s translucid contracts enable modular reasoning about control effects of observers of an event and observers of its superevents, independent of observers and their execution orders. This is sound because each conforming observer of an event has the same control effects as the translucid contract of the event and *Ptolemy*$_{\mathbb{S}}$'s refining relation ensures that the contract of an event refines the control effects of the contract of its superevent. Control effects are specified by program expressions in translucid contracts.

In *Ptolemy*$_{\mathbb{S}}$, the assertion Ψ could be verified using the translucid contract of `AndEv` and especially its assumes block, on lines 21–25 of Figure 4. The program expression **next**.**invoke**(), on line 22, guarantees that each observer of `AndEv` includes the invoke expression in their implementations; and the refining relation ensures that each observer of superevents of `AndEv` contain the invoke expression in their implementations too. This means that the invoke expression in the implementation of `evaluator` or `tracer` in χ_2 cannot go missing or otherwise these observers will not be conforming to their translucid contracts. This in turn means that all the observers in the chain χ_2 including the event body at the end of the chain are invoked and executed.

6.1 Control Interference of Subjects and Observers

Rinard *et al.* [25] classify the control interactions of a subject and observer of an event into four categories: *(i)* augmentation, *(ii)* narrowing, *(iii)* replacement and *(iv)* combination. These categories are concerned about the number of invoke expressions and their executions in an implementation of an observer. An augmentation observer executes its invoke expression exactly once, narrowing executes it at most once, replacement does not execute any invoke expressions and a combination observer executes its invoke expression zero or more times in its implementation.

Ptolemy$_{\mathbb{S}}$'s translucid contracts allow modular reasoning about control interference category of interactions of subjects and observers of an event, independent of observers of the event and observers of its superevents. To reason about the control interference of subjects and observers of an event, one uses the translucid contract of the event to decide about the the number of times invoke expressions of the translucid contract may execute. An invoke expression surrounded by an if conditional executes at most once, whereas an invoke expression surrounded by a loop may execute zero times or more. Otherwise an invoke expression executes exactly once. This is sound, because the structural refinement of the conformance relation requires each observer of an event to have the same control effects as its translucid contracts especially regarding the number of invoke expressions in its implementation. Also the refining relation ensures that the control effects of observers of an event refine the control effects of observers of its superevents.

[4] *Ptolemy*$_{\mathbb{S}}$' core does not support throwing or handling of exceptions [6].

Augementation interactions and observers To illustrate the augmentation interaction, consider the observer `Evaluator` and subject `ASTVisitor` of the event `AndEv`. Using only the translucid contract of `AndEv`, on lines 20–26 of Figure 4, one can conclude that subjects and observers of `AndEv` have an augmentation interaction, in which `Evaluator` augments the behavior of its subject, i.e. `Evaluator` is an augmentation observer. This is because, the assumes block of the contract of `AndEv` contains an invoke expression, on line 22, which is not surrounded by any conditionals or loops. This in turn means that the conforming observer `Evaluator` has only one invoke expression in its implementation which executes exactly once. For observers `Checker` and `Tracer` of superevents `BinEv` and `ExpEv` of `AndEv`, the refining relation ensures that they also have only one invoke expressions in their implementations and thus they are augmentation observers too.

For an event with augmentation interactions and observers, one can conclude that upon announcement of the event all observers of the event and observers of its superevent including the event body execute and no execution is skipped, similar to assertion Ψ.

Replacement interactions and observers To illustrate the replacement interaction, consider the translucid contract of `AndEv` with its translucid contract in Figure 4, but without its invoke expression. Using this translucid contract one can conclude that subjects and observers of `AndEv` have a replacement interaction, in which `Evaluator` replaces the body of its announce expression in a subject, i.e. `Evaluator` is a replacement observer. To structurally refines its contract, `Evaluator` cannot have any invoke expression in its implementation. The refining relation ensures that superevents `BinEv` and `ExpEv` cannot have invoke expressions in their contracts either and thus observers `Checker` and `Tracer` are replacement observers too.

For an event with replacement observers, one can conclude that upon announcement of the event the first observer of the event or its superevents executes and executions of the rest of the observers including the event body are skipped. This is because none of the observers have an invoke expression in their implementations.

7. Discussion

Implementation To prove the feasibility of the proposed language, we implemented *Ptolemy*$_{\mathbb{S}}$'s compiler on top of Ptolemy's compiler [16] which itself is an extension of the OpenJDK Java compiler. To the previous compiler, we added: translucid contracts, static structural refinement, static event specification inheritance, runtime assertion checking of preconditions and postconditions of contracts and their specification expressions, and a non-decreasing execution order of observers of an event and its superevents. Compared to Ptolemy's compiler, maintaining separate lists for observers of separate events, rather than a single global list, simplified implementation of event announcement and handling especially with dynamic (un)registration of observers.

Limitation A non-decreasing relation among observers of an event and its superevent(s) limits execution order of observers and could require a programmer to co-design the event subtyping hierarchy of a program and execution order of their observers. Without such a co-design, there could be some execution orders of observers that may not be allowed by a specific event subtyping hierarchy. For example, with the event hierarchy in our expression language example, observer `evaluator` always runs before `checker`. Placement of invoke expressions in observers play an important role in the functionality of a system. For example, although `evaluator` runs before `checker`, an expression is not evaluated unless it is first type checked. This is enforced because `evaluator` invokes the handler chain before evaluating.

8. Related Work

Modular type checking Previous work on join point types (JPT) [15], join point interfaces (JPI) [14], and Ptolemy's typed events [16], enable modular type checking of subjects and observers of subtyping event types. EventJava [9] extends Java with events and event correlation in distributed settings and Escala [4] extends Scala with explicitly declared events as members of classes. However, previous works are not concerned with modular reasoning about behaviors and control effects of subjects and observers of events using specification of subtyping event types.

Modular reasoning Previous work on MAO [21], EffectiveAdvice [32], MRI [33] and the work of Khatchadourian *et al.* [19] enables modular reasoning, however, it does not use explicit interfaces among subjects and observers and thus is not concerned about their subtyping. Previous work on crosscutting programming interfaces (XPI) [3], crosscutting programming interfaces with design rules (XPIDR) [20] and open modules [2] enables modular reasoning using explicit interfaces, however, it is not concerned about subtyping of these interfaces. Translucid contracts [5–7] proposes event type specifications to enable modular reasoning, however it is not concerned with event subtyping.

Modular reasoning about dynamic dispatch Supertype abstraction [34] enables modular reasoning about invocation of a dynamically dispatched method in the presence of class subtyping [34], relying on a refinement relation among blackbox contracts of a supertype and its subtypes [23, 35]. *Ptolemy*$_{\mathbb{S}}$'s refining of event contracts is the inverse of the refinement in supertype abstraction and extends it to greybox contracts with control effects. Refinement in supertype abstraction relies on known links among method invocations and method names, whereas in *Ptolemy*$_{\mathbb{S}}$ there is no link among subjects and observers of an event [6, 18]. Subjects and observers do not know about each other and only know their event. Unlike a method invocation which invokes exactly one method, announcement of an event in *Ptolemy*$_{\mathbb{S}}$ by a subject could invoke zero or more observers of the event and observers of its superevents where all these observers and the subject must conform to their event specifications. The challenge in supertype abstraction is modular reasoning about a method invocation independent of the dynamic types of its receiver, whereas in *Ptolemy*$_{\mathbb{S}}$ the challenge is tractable reasoning about announcement and handling of an event, independent of its observers, observers of its superevents and their execution orders, while allowing reuse of events.

9. Conclusion and Future Work

In this work, we identified two problems of combinatorial reasoning and behavior invariance in modular reasoning about subjects and observers in the presence of event subtyping, that threaten tractability of reasoning and reuse of events. We proposed a refining relation among greybox event specifications of events in a subtyping hierarchy, a non-decreasing relation on execution orders of their observers, and a conformance relation among subjects and observers of an event and their translucid contract to solve these problems in the context of a new language design called *Ptolemy*$_{\mathbb{S}}$. We discussed *Ptolemy*$_{\mathbb{S}}$'s modular reasoning and showed its applicability to other AspectJ-like [1] event-based systems such as join point types [15]. Future work includes performing a large experimental study to further investigate benefits of *Ptolemy*$_{\mathbb{S}}$'s event model and its modular reasoning.

References

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

[2] Aldrich, J.: Open Modules: Modular reasoning about advice. In: ECOOP'05

[3] Sullivan, K.J., Griswold, W.G., Rajan, H., Song, Y., Cai, Y., Shonle, M., Tewari, N.: Modular aspect-oriented design with XPIs. TOSEM'11 **20**(2)

[4] Gasiunas, V., Satabin, L., Mezini, M., nez, A.N., Noyé, J.: EScala: modular event-driven object interactions in Scala. In: AOSD'11

[5] Bagherzadeh, M., Rajan, H., Leavens, G.T., Mooney, S.: Translucid contracts: Expressive specification and modular verification for aspect-oriented interfaces. In: AOSD'11

[6] Bagherzadeh, M., Rajan, H., Darvish, A.: On exceptions, events and observer chains. In: AOSD'13

[7] Sánchez, J., Leavens, G.: Separating obligations of subjects and handlers for more flexible event type verification. In: Software Composition'13

[8] Hoffman, K., Eugster, P.: Bridging Java and AspectJ through explicit join points. In: PPPJ'07

[9] Eugster, P., Jayaram, K.R.: EventJava: An extension of Java for event correlation. In: ECOOP'09

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

[11] Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. ICSE'05

[12] Clifton, C., Leavens, G.T.: A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-23, Iowa State University (2005)

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

[14] Bodden, E., Tanter, E., Inostroza, M.: Joint point interfaces for safe and flexible decoupling of aspects. TOSEM'13

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

[16] Fernando, R., Dyer, R., Rajan, H.: Event type polymorhphism. In: FOAL'12

[17] Xu, J., Rajan, H., Sullivan, K.: Understanding aspects via implicit invocation. In: ASE '04

[18] Dingel, J., Garlan, D., Jha, S., Notkin, D.: Towards a formal treatment of implicit invocation using rely/guarantee reasoning. Formal Asp. Comput. '98 **10**(3)

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

[20] Rebelo, H., Leavens, G.T., Lima, R.M.F., Borba, P., Ribeiro, M.: Modular aspect-oriented design rule enforcement with XPIDRs. In: FOAL'13

[21] Clifton, C., Leavens, G., Noble, J.: MAO: Ownership and effects for more effective reasoning about aspects. In: ECOOP'07

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

[23] Leavens, G.T., Naumann, D.A.: Behavioral subtyping, specification inheritance, and modular reasoning. Technical Report CS-TR-13-03, Computer Science, University of Central Florida (2013)

[24] Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM'83 **26**(1)

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

[26] Bagherzadeh, M., Dyer, R., Fernando, R.D., Rajan, H., Sánchez, J., Leavens, G.T.: Modular reasoning in the presence of event subtyping. Technical Report 14-02b, Iowa State University, Department of Computer Science (2014)

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

[28] Dhara, K.K., Leavens, G.T.: Forcing behavioral subtyping through specification inheritance. In: ICSE'97

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

[30] Abadi, M., Leino, K.: A logic of object-oriented programs. In: TAPSOFT'97

[31] Boer, F.: A WP-calculus for OO. In: Foundations of Software Science and Computation Structures'99

[32] Oliveira, B.C.d.S., Schrijvers, T., Cook, W.R.: EffectiveAdvice: Disciplined advice with explicit effects. In: AOSD'10

[33] Oliveira, B.c.d.s., Schrijvers, T., Cook, W.r.: MRI: Modular reasoning about interference in incremental programming. J. Funct. Program.'12 **22**(6)

[34] Leavens, G.T., Weihl, W.E.: Specification and verification of object-oriented programs using supertype abstraction. Acta Informatica'95

[35] Liskov, B.H., Wing, J.M.: A behavioral notion of subtyping. TOPLAS'94 **16**(6)

[36] Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation '94 **115**(1)

A. Dynamic Semantics

In this section, we present a substitution-based small-step operational semantics for *Ptolemy*$_{\mathbb{S}}$ with special focus on announcing and handling of events in an event inheritance hierarchy and the non-decreasing relation on execution order of their observers.

A.1 Dynamic Semantic Objects

Added syntax:
$e ::= loc \mid \texttt{evalpost}\ e\ q$
$\quad \mid \texttt{NPE} \mid \texttt{CCE} \mid \texttt{TCE}$
$\quad$ **where** $loc \in \mathscr{L}$, a set of locations

Evaluation contexts:
$\mathbb{E} ::= - \mid \mathbb{E}\,.\,m(e\ldots) \mid v\,.\,m(v\ldots\mathbb{E}e\ldots) \mid \mathbb{E}\,.\,f \mid \mathbb{E}\,.\,f{=}e$
$\quad \mid \texttt{if}\ (\mathbb{E})\ \{\ e\ \}\ \texttt{else}\ \{\ e\ \} \mid \texttt{cast}\ c\ \mathbb{E} \mid t\ var{=}\mathbb{E};\ e$
$\quad \mid \texttt{announce}\ (v\ldots\mathbb{E}e\ldots)\ \{e\} \mid \texttt{invoke}(\mathbb{E})$
$\quad \mid \texttt{register}(\mathbb{E}) \mid \texttt{unregister}(\mathbb{E})$
$\quad \mid \texttt{refining requires}\ \mathbb{E}\ \texttt{ensures}\ q$

Evaluation relation: $\hookrightarrow$: $\langle e,S,\Pi,A\rangle \rightarrow \langle e',S',\Pi',A'\rangle$

Domains:

$\Sigma ::= \langle e,S,\Pi,A\rangle$	"Configurations"
$S ::= \{loc_k \mapsto sv_k\}_{k\in K}$,	"Stores"
$v ::= \texttt{null} \mid loc$	"Values"
$sv ::= or \mid ec$	"Storable Values"
$or ::= [c\,.\,F]$	"Object Records"
$F ::= \{f_k \mapsto v_k\}_{k\in K}$,	"Field Maps"
$\rho ::= \{var \mapsto v_k\}_{k\in K}$,	"Environments"
$ec ::= \texttt{eClosure}(H,e,\rho)$	"Event Closure"
$H ::= h + H \mid \bullet$	"Handler Records List"
$h ::= \langle loc,m\rangle$	"Handler Record"
$A ::= \{ev_k \mapsto O_k\}$	"Active Objects Map"
$O ::= loc + O \mid \bullet$	"Active Objects List"
where K is finite	

Figure 12. Added syntax, evaluation contexts and configuration.

Ptolemy$_{\mathbb{S}}$'s operational semantics relies on few additional expressions that are not part of its surface syntax, as shown in Figure 12, including loc to represent the locations in the store and

evalpost $e\ q$ to check that the expression e satisfies the postcondition q. *Ptolemy*$_{\mathbb{S}}$ also uses three exceptions to represent dereferencing null references, i.e. **NPE**, runtime cast exceptions, i.e. **CCE**, and violations of translucid contracts, i.e. **TCE**. In *Ptolemy*$_{\mathbb{S}}$'s core semantics, exceptions are terminal states [13]. Figure 12 also shows the evaluation contexts used in *Ptolemy*$_{\mathbb{S}}$'s dynamic semantics. An evaluation context $\mathbb{E}$ specifies the evaluation order and the position in an expression where the evaluation is happening. *Ptolemy*$_{\mathbb{S}}$ uses a left-most inner-most call-by-value evaluation policy.

Ptolemy$_{\mathbb{S}}$'s operational semantics, transitions from one configuration to another. A configuration Σ, in Figure 12 contains an expression e, store S, store typing Π and a mapping A from events ev to their ordered list of observers O. A store maps locations to storable values sv which themselves are either an object record or or an event closure ec. An object record has a class name c and a map F from fields to their values. An event closure $\mathtt{eClosure}(H,e,\rho)$ contains an ordered list of observer handlers H, an expression e and an environment ρ for running e. An observer handler method h contains a location loc that points to its observer object and a its method handler name m. A value v is either a location loc or **null**. A store typing is maintained and updated by the dynamic rules only to be used in the soundness proof.

A.2 Dynamic Semantic Rules

Evaluation relation: $\boxed{\hookrightarrow: \langle e,S,\Pi,A\rangle \to \langle e',S',\Pi',A'\rangle}$

(ANNOUNCE)
$$\frac{\begin{array}{c}(c\ \mathtt{event}\ ev\ \mathtt{extends}\ ev'\{(t\ var)^*\ contract_{ev}\}) \in CT \\ loc \notin dom(S) \\ H = handlersOf(ev) \qquad \rho = \{var_i \mapsto v_i \mid var_i \in var^* \wedge v_i \in v^*\} \\ S' = S \uplus (loc \mapsto \mathtt{eClosure}(H,e,\rho)) \qquad \Pi' = \Pi \uplus (loc : \mathtt{thunk}\ ev)\end{array}}{\langle \mathbb{E}[\mathtt{announce}\ ev\ (v^*)\ \{e\}],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[loc.\mathtt{invoke}()],S',\Pi',A\rangle}$$

(INVOKEDONE)
$$\frac{\mathtt{eClosure}(\bullet,e,\rho) = S(loc)}{\langle \mathbb{E}[loc.\mathtt{invoke}()],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[e],S,A,\Pi\rangle}$$

(INVOKE)
$$\frac{\begin{array}{c}\mathtt{eClosure}(\langle loc',m\rangle + H,e,\rho) = S(loc) \\ [c\,.\,F'] = S(loc') \qquad (c_2, t\ m(t_1\ var_1)\{e'\}) = methodBody(c,m) \\ e'' = [loc_1/var_1, loc'/\mathtt{this}]e' \qquad loc_1 \notin dom(S) \\ S' = S \uplus (loc_1 \mapsto \mathtt{eClosure}(H,e,\rho)) \qquad \Pi' = \Pi \uplus (loc_1 : \Pi(loc))\end{array}}{\langle \mathbb{E}[loc.\mathtt{invoke}()],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[e''],S',\Pi',A\rangle}$$

(REGISTER)
$$\frac{\forall ev \in eventsOf(loc)\ .\ A'[ev] = A[ev] + loc}{\langle \mathbb{E}[\mathtt{register}(loc)],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[loc],S,\Pi,A'\rangle}$$

(UNREGISTER)
$$\frac{\forall ev \in eventsOf(loc)\ .\ A'[ev] = A[ev] - loc}{\langle \mathbb{E}[\mathtt{unregister}(loc)],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[loc],S,\Pi,A'\rangle}$$

(REFINING)
$$\frac{n \neq 0}{\begin{array}{c}\langle \mathbb{E}[\mathtt{refining\ requires}\ n\ \mathtt{ensures}\ q\ \{e\}],S,\Pi,A\rangle \hookrightarrow \\ \langle \mathbb{E}[\mathtt{evalpost}\ e\ q],S,\Pi,A\rangle\end{array}}$$

(EVALPOST)
$$\frac{n \neq 0}{\langle \mathbb{E}[\mathtt{evalpost}\ v\ n],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[v],S,\Pi,A\rangle}$$

(ECGET)
$$\frac{\mathtt{eClosure}(H,e,\rho) = S(loc) \qquad v = \rho(f)}{\langle \mathbb{E}[loc.f],S,\Pi,A\rangle \hookrightarrow \langle \mathbb{E}[v],S,\Pi,A\rangle}$$

Figure 13. Select rules for *Ptolemy*$_{\mathbb{S}}$'s dynamic semantics, based on [13].

$$handlersOf(\mathtt{Event}) = \bullet$$

$$\frac{(c\ \mathtt{event}\ ev\ \mathtt{extends}\ ev'\{form^*\ contract_{ev}\}) \in CT}{handlersOf(ev) = hbind(ev,S,A[ev]) \oplus handlersOf(ev')}$$

$$hbind(ev,S,\bullet) = \bullet$$

$$\frac{[c.F] = S(loc) \qquad B = bindingsOf(c)}{hbind(ev,S,loc + A[ev]) = match(B,ev,S,loc) \oplus hbind(ev,S,A[ev])}$$

$$bindingsOf(\mathtt{Object}) = \bullet$$

$$\frac{(\mathtt{class}\ c\ \mathtt{extends}\ d\ \{form^*\ meth^*\ binding^*\}) \in CT}{bindingsOf(c) = binding^* \oplus bindingsOf(d)}$$

$$match(\bullet,ev,S,loc) = \bullet$$

$$match((\mathtt{when}\ ev\ \mathtt{do}\ m) + B,ev,S,loc) = (\langle loc,m\rangle + match(B,ev,S,loc))$$

$$\frac{[c.F] = S(loc) \qquad B = bindingsOf(c)}{eventsOf(loc) = registeredFor(loc,B)} \qquad registeredFor(loc,\bullet) = \bullet$$

$$registeredFor(loc,(\mathtt{when}\ ev\ \mathtt{do}\ m) + B) = ev \oplus registeredFor(loc,B)$$

Figure 14. Select auxiliary functions for *Ptolemy*$_{\mathbb{S}}$'s dynamic semantics, based on [6, 13].

Figure 13 shows the dynamic semantic rules for *Ptolemy*$_{\mathbb{S}}$-specific expressions. In *Ptolemy*$_{\mathbb{S}}$ a subject announces an event using an announce expression, observers (un)register for the event using (un)register expressions, and invoke each other using invoke expressions.

The rule (ANNOUNCE), says that upon announcement of an event ev an event closure $\mathtt{eClosure}(H,e,\rho)$ is constructed that contains the list (chain) of observer handler methods of the event and the observer handler methods of its superevent, in H, the event body e and an environment mapping context variables var^* of the event to their values v^*, in ρ. The list H is constructed using the auxiliary function *handlersOf*, in Figure 14. The function *handlersOf* first computes the list of observer handler methods of the event ev, using *hbind*, and concatenates it to the handlers of the superevents ev' until the event **Event** is reached. This in turn ensures that the observer handler methods of the event ev appear before the observer handler methods of its superevent ev' in the list of observer handler methods H, according to the non-decreasing relation. The event **Event** has no observers since is not part of *Ptolemy*$_{\mathbb{S}}$'s surface syntax and observers can not register for or handle it. The concatenate operator $\oplus$ ignores empty $\bullet$ elements. The function *hbind* binds the observer loc, in the beginning of the $A[ev]$, to observer handler method m, using the auxiliary function *match* and concatenates it to the bindings for the rest of $A[ev]$. After construction, the event closure is mapped to a fresh location loc and the execution of the chain of observer handler methods starts using the invoke expression, i.e. $loc.\mathtt{invoke}()$.

(ANNOUNCE) also updates the store typing environment Π with a new mapping from the location loc to the type **thunk** ev of the event closure it points to. Recall that thunk types mark event closure types. The operator $\uplus$ is an overriding union operator.

When executing the chain of observer handler methods H, (INVOKEDONE) executes the event body e, if there are no more observer handler methods in the list, i.e. H is $\bullet$. Otherwise, (INVOKE), executes the next observer handler method m of the observer instance loc'. The rule (INVOKE) finds the body e' of the method m, using the auxiliary function *methodBody*, replaces its parameter loc_1 and **this** with their values var_1 and loc' and executes the result expression e''. The auxiliary function *methodBody* emulates dynamic dispatch at runtime. After the execution of the observer handler method at the beginning of the list H, the event closure is updated to reflect the execution of the observer and the updated event closure is stored at a fresh location loc_1. (INVOKE) also updates the store typing environment Π with a mapping between the location loc_1 of the new event closure and its type.

Announce and invoke expressions execute observers that are registered for the event. The rule (REGISTER) adds the location *loc* of an observer instance to the list $A[ev]$ of active objects for the event *ev*. An observer with multiple binding declarations is register for all the events named in its binding declarations. The auxiliary function *eventsOf* computes all events an observer is registered for by computing all its bindings, using the auxiliary function *bindingsOf*. Similarly, the rule (UNREGISTER) removes an observer instance from the list of active objects for all events it was registered to.

A refining expression claims that its body satisfies the precondition and postcondition of its specification. The rule (REFINING) ensures that the body e of a refining expression actually refines its specification. If the precondition n is satisfied, i.e. $n \neq 0$, the rule checks for the satisfaction of the postcondition q using an **evalpost** expression. The rule (EVALPOST) ensures that the expression e satisfies the postcondition q. Violation of the precondition or postcondition of a refining expression causes throwing a **TCE**, in the rules (X-REFINING) and (X-EVALPOST) in Figure 15, and termination of the program. In *Ptolemy*$_{\mathbb{S}}$'s core semantics, exceptional states are terminal states. Figure 15 shows the dynamic semantics of *Ptolemy*$_{\mathbb{S}}$'s exceptional termination.

(X-REFINING)
$$\frac{n == 0}{\langle \mathbb{E}[\mathbf{refining\ requires}\ n\ \mathbf{ensures}\ q\ \{e\}], S, \Pi, A\rangle \hookrightarrow \langle \mathbf{TCE}, S, \Pi, A\rangle}$$

(X-REGISTER)
$$\langle \mathbb{E}[\mathbf{register}(\mathbf{null})], S, \Pi, A\rangle \hookrightarrow \langle \mathbf{NPE}, S, \Pi, A\rangle$$

(X-UNREGISTER)
$$\langle \mathbb{E}[\mathbf{unregister}(\mathbf{null})], S, \Pi, A\rangle \hookrightarrow \langle \mathbf{NPE}, S, \Pi, A\rangle$$

(X-EVALPOST)
$$\frac{n == 0}{\langle \mathbb{E}[\mathbf{evalpost}\ v\ n], S, \Pi, A\rangle \hookrightarrow \langle \mathbf{TCE}, S, \Pi, A\rangle}$$

(X-CAST)
$$\frac{[c\,.\,F] = S(loc) \qquad c \not\preccurlyeq t}{\langle \mathbb{E}[\mathbf{cast}\ t\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbf{CCE}, S, \Pi, A\rangle}$$

Figure 15. *Ptolemy*$_{\mathbb{S}}$'s exceptional dynamic semantics.

Ptolemy$_{\mathbb{S}}$ also supports standard object-oriented expressions for object creation, getting and setting the value of a field, if conditionals, etc. Their semantics can be found in Section E.

B. Type Checking

In this section, we discuss *Ptolemy*$_{\mathbb{S}}$'s static semantics with the focus on event subtyping, the refining relation among greybox event specifications and the non-decreasing relation.

B.1 Type Attributes

Figure 16 defines the type attributes used in *Ptolemy*$_{\mathbb{S}}$'s typing rules. The type attribute OK shows that a higher level declaration type checks, whereas OK in c shows type checking in the context of a class c. Other type attributes **var** t and **exp** t show variables and expressions of type t, respectively. Variable and store typing environments Γ and Π, respectively, map variables and locations

$\theta ::=$	"type attributes"
OK	"program/top-level decl."
$\mid$ OK in c	"method, binding"
$\mid$ **var** t	"var/formal/field"
$\mid$ **exp** t	"expression"
$t ::= c \mid \text{int} \mid \text{bool}$	"types"
$\Gamma ::= \{var : t\}$	"variable typing environment"
$\Pi ::= \{loc : t\}$	"store typing environment"
$\Gamma, \Pi \vdash e : \theta$	"typing judgement"

Figure 16. Type attributes, based on [13]

to their types. The typing judgment $\Gamma, \Pi \vdash e : \theta$ says that in the variable typing environment Γ and the store typing environment Π, the expression e has the type θ. *Ptolemy*$_{\mathbb{S}}$'s type checking rules use a fixed class table *CT*, which is a set of program's class and event type declarations. Top-level names in a program are distinct and inheritance relations on classes and events types are acyclic.

B.2 Static Semantics Rules

Figure 17 shows several typing rules for *Ptolemy*$_{\mathbb{S}}$. The rest of *Ptolemy*$_{\mathbb{S}}$'s typing rules, which are mostly standard object oriented rules could be found in Section B.

The rule (T-EVENT) type checks the declaration of an event *ev*. Since *ev* extends another event ev', the rule ensures that *ev* is a valid subevent of ev', i.e. $ev \lll: ev'$, and its translucid contract refines the translucid contract of ev', i.e. $contract_{ev'} \trianglelefteq contract_{ev}$. The refinement of the translucid contract of ev' by the contract of *ev* is statically guaranteed by *Ptolemy*$_{\mathbb{S}}$'s specification inheritance. The rule (T-EVENT) also checks, using the auxiliary function *isClass*, that the return type and types of context variables of *ev* are valid class types. Figure 18 shows the auxiliary functions used in *Ptolemy*$_{\mathbb{S}}$'s typing rules. The auxiliary function *isClass* simply ensures that its parameter is a class declared in the class table *CT*.

(T-SUBEVENT) checks that an event *ev* is a valid subtype of event ev', regarding both width and depth subtyping. Width subtyping allows *ev* to declare context variables in addition to the context it inherits from its superevent ev', i.e. $contextsOf(ev') \subseteq contextsOf(ev)$. The auxiliary function *contextsOf* returns all the context variables of an event along with their types, including context inherited from all of its superevents. Depth subtyping allows *ev* to redeclare a context variable of its superevent ev'. To redeclare a context variable var_i of type t'_i, the redeclaring context variable must have the same name var_i and its type t_i must be a subtype of t'_i, i.e. $t_i \preccurlyeq t'_i$. Similar to class subtyping, event subtyping is a reflexive, transitive relation declared among event types, with a root event type of **Event**.

(T-SUBEVENT) also ensures that the return type of an event *ev* is a *supertype* of the return type of its superevent ev'. This is necessary for the non-decreasing relation on observers of an event and its superevent, shown in Figure 5, which ensures that an observer of an event runs before an observer of its superevents. The auxiliary function *returnType* returns the return type of an event.

(T-ANNOUNCE) type checks an announce expression. It ensures that the type of a parameter expression e_i is a subtype of its corresponding context variable var_i, i.e. $t'_i \preccurlyeq t_i$. Recall that an event can inherit context variables from its superevents and the announce expression must provide values for all context variables of the event.

(T-ANNOUNCE) also ensures that the type of the event body e' is the same as the return type of the top event of its event. The top event of an event in an inheritance hierarchy is the superevent of the event right before the root event **Event**. For example, in Figure 2, the event `ExpEv` is the top event of `AndEv`. The auxiliary function *topEvent* returns the top event of an event. The relation between the return type of the event body and the the return type of its top event

(T-EVENT)
$$\frac{\begin{array}{c}(c'\ \texttt{event}\ ev'\ \texttt{extends}\ ev''\ \{(t'\ var')^*\ contract_{ev'}\}) \in CT \\ \Gamma,\Pi \vdash contract_{ev'} \trianglelefteq contract_{ev} \\ \vdash ev \ll: ev' \quad isClass(c) \quad \forall t_i \in t^*\ .\ isClass(t_i)\end{array}}{\vdash c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{(t\ var)^*\ contract_{ev}\} : \text{OK}}$$

(T-SUBEVENT)
$$\frac{\begin{array}{c}contextsOf(ev') \subseteq contextsOf(ev) \\ (t\ var)^* = contextsOf(ev) \quad (t'\ var')^* = contextsOf(ev') \\ \forall\ (t_i\ var_i) \in (t\ var)^*,\ (t'_i\ var_i) \in (t'\ var')^*\ .\ t_i \preccurlyeq t'_i \\ returnType(ev') \preccurlyeq returnType(ev)\end{array}}{\vdash ev \ll: ev'}$$

(T-ANNOUNCE)
$$\frac{\begin{array}{c}(t\ var)^* = contextsOf(ev) \\ \forall e_i \in e^*,\ (t_i\ var_i) \in (t\ var)^*\ .\ \Gamma,\Pi \vdash e_i : \texttt{exp}\ t'_i \wedge t'_i \preccurlyeq t_i \\ c''\ \texttt{event}\ ev'\texttt{extends}\ \texttt{Event}\{\} = topEvent(ev) \\ c = returnType(ev) \quad \Gamma,\Pi \vdash e' : \texttt{exp}\ c''\end{array}}{\Gamma,\Pi \vdash \texttt{announce}\ ev\ (e^*)\ \{e'\} : \texttt{exp}\ c}$$

(T-BINDING)
$$\frac{\begin{array}{c}(c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{form^*\ contract_{ev}\}) \in CT \\ contract_{ev} = \texttt{requires}\ p\ \texttt{assumes}\ \{se\}\ \texttt{ensures}\ q \\ (c\ m\ (\texttt{thunk}\ ev\ var)\ \{e\}) = methodBody(c', m) \quad se \trianglelefteq e\end{array}}{\vdash \texttt{when}\ ev\ \texttt{do}\ m : \text{OK in } c'}$$

(T-INVOKE)
$$\frac{\begin{array}{c}c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{form^*\ \ contract_{ev}\} \in CT \\ \Gamma,\Pi \vdash e : \texttt{exp}\ \texttt{thunk}\ ev\end{array}}{\Gamma,\Pi \vdash e.\texttt{invoke}() : \texttt{exp}\ c}$$

(T-REGISTER)
$$\frac{\Gamma,\Pi \vdash e : \texttt{exp}\ t}{\Gamma,\Pi \vdash \texttt{register}(e) : \texttt{exp}\ t}$$

(T-UNEGISTER)
$$\frac{\Gamma,\Pi \vdash e : \texttt{exp}\ t}{\Gamma,\Pi \vdash \texttt{unregister}(e) : \texttt{exp}\ t}$$

(T-EVALPOST)
$$\frac{\Gamma,\Pi \vdash e : \texttt{exp}\ t \quad \Gamma,\Pi \vdash q : \texttt{exp}\ t_2}{\Gamma,\Pi \vdash \texttt{evalpost}\ e\ q : \texttt{exp}\ t}$$

(T-SPEC)
$$\frac{\Gamma,\Pi \vdash p : \texttt{exp}\ t_1 \quad \Gamma,\Pi \vdash q : \texttt{exp}\ t_2}{\Gamma,\Pi \vdash \texttt{requires}\ p\ \texttt{ensures}\ q\ : \texttt{exp}\ \bot}$$

(T-REFINING)
$$\frac{\begin{array}{c}spec = \texttt{requires}\ p\ \texttt{ensures}\ q \\ \Gamma,\Pi \vdash spec : \texttt{exp}\bot \quad \Gamma,\Pi \vdash e : \texttt{exp}\ t\end{array}}{\Gamma,\Pi \vdash \texttt{refining}\ spec\ \{e\} : \texttt{exp}\ t}$$

(T-PROGRAM)
$$\frac{\forall decl \in decl^*\ .\ \vdash decl : \text{OK} \quad \vdash e : \texttt{exp}\ t}{\vdash decl^*\ e : \texttt{exp}\ t}$$

(T-CLASS)
$$\frac{\begin{array}{c}\forall meth \in meth^*\ .\ \vdash meth : \text{OK in } c \\ \forall binding \in binding^*\ .\ \vdash binding : \text{OK in } c \\ isClass(d) \quad \forall (t\ f) \in form^*\ .\ isClass(t) \wedge f \notin dom(fieldsOf(d))\end{array}}{\vdash \texttt{class}\ c\ \texttt{extends}\ d\ \{form^*\ meth^*\ binding^*\} : \text{OK}}$$

Figure 17. Select typing rules for *Ptolemy*$_{\mathbb{S}}$ [6, 16].

is necessary for the non-decreasing relation in which the event body runs as the last observer, as in Figure 5.

(T-BINDING) type checks a binding declaration. It ensures that the body e of the observer handler method m refines the assumes block se of the translucid contract of its event ev, i.e. $se \trianglelefteq e$, as defined in Figure 9. The auxiliary function *methodBody* returns the body of a method of a class defined in the class table CT. The rule also ensures that the return type of the observer handler method m is the same as the the return type of the event.

(T-INVOKE) type checks an invoke expression. The invoke expression invokes the next observer in the chain of observers. The chain of observers is included in the event closure receiver object e. The rule ensures that the event closure of an event ev is of type **thunk** ev. A thunk type marks the type of an event closure. The type of an invoke expression is the same as the return type c of its event ev. This is sound because the non-decreasing relation ensures that observers of an event run before observers of its superevent.

(T-REGISTER) type checks a register expression. The type of a register expression is the same as the type of its parameter. The rule (T-UNREGISTER), similarly type checks an unregister expression. (T-SPEC) type checks a specification expression. A specification expression has the bottom type $\bot$ which is the subtype of any other type. The bottom type in turn allows a specification expression to be refined by a refining expression of any type. The rule ensures that the precondition p and the postcondition q of the specification expressions type check too. (T-REFINING) type checks a refining expression. The rule simply says that the type of a refining expression is the same as the type of its body e. (T-EVALPOST) type checks an evalpost expression and says that the type of an evalpost expression is the same as the type of its body e. An evalpost expression, is used to check that an expression e satisfies a postcondition q, as discussed in Section A.

$$\frac{\begin{array}{c}(c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{(t\ var)^*\ contract_{ev}\}) \in CT \\ (t'\ var')^* = contextsOf(ev')\end{array}}{contextsOf(ev) = (t'\ var')^* \oplus (t\ var)^*}$$

$$contextsOf(\texttt{Event}) = \bullet$$

$$\frac{(c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{form^*\ contract_{ev}\}) \in CT}{returnType(ev) = c}$$

$$\frac{(c\ \texttt{event}\ ev\ \texttt{extends}\ ev'\ \{form^*\ contract_{ev}\}) \in CT}{isEvent(ev)}$$

$$\frac{\texttt{class}\ c\ \texttt{extends}\ d\{form^*\ meth^*\ binding^*\} \in CT}{isClass(c)}$$

$$\frac{t = \texttt{thunk}\ ev}{isThunkType(t)} \qquad \frac{isClass(t) \vee isThunkType(t)}{isType(t)}$$

$$\frac{\texttt{class}\ c\ \texttt{extends}\ d\{(t\ var)^*\ meth^*\ binding^*\} \in CT}{fieldsOf(c) = (var : t)^*}$$

$$\frac{\begin{array}{c}\texttt{class}\ c\ \texttt{extends}\ d\{form^*\ meth^*\ binding^*\} \in CT \\ (c''\ m\ (t\ var)^*\ \{e\}) \in meth^*\end{array}}{methodBody(c,m) = (c''\ m\ (t\ var)^*\ \{e\})}$$

$$\frac{\begin{array}{c}\texttt{class}\ c\ \texttt{extends}\ d\{form^*\ meth^*\ binding^*\} \in CT \\ (c''\ m\ (t\ var)^*\ \{e\}) \notin meth^*\end{array}}{methodBody(c,m) = methodBody(d,m)}$$

Figure 18. Select auxiliary functions for *Ptolemy*$_{\mathbb{S}}$'s typing rules, based on [6, 13].

B.3 Soundness of Type System

THEOREM B.1. *(Soundness of Ptolemy$_{\mathbb{S}}$'s Semantics) Ptolemy$_{\mathbb{S}}$'s semantics is sound regarding its progress and preservation [36].*

The proof follows standard progress and preservation arguments. Full proof of the theorem can be found in Section E.

C. More on Applicability

Despite their differences [14], event announcement and handling and event subtyping models of join point interface [14] and join point types [15] are similar. This in turn allows use of syntax and refinement rules similar to join point types for join point interfaces.

Figure 19 shows parts of the boolean expression example rewritten using join point interfaces. Unlike join point types or *Ptolemy*$_{\mathbb{S}}$'s event type declarations which are similar to type declarations, join point interfaces are declared similar to method signatures. Also context variables of join point interfaces are explicitly named in aspects and their proceed statements, unlike join point types or *Ptolemy*$_{\mathbb{S}}$ that use join point instances and event closures. For example, line 31 of Figure 19, shows the declaration of join point interface `AndEv` and explicit naming of context variables `node`, `left` and `right` in the aspect `Evaluator`, on line 44, and its proceed statement, on line 45.

```
1 /* join point interfaces */
2 /*@ requires node != null;
3   @ model_program {
4   @  proceed(node);
5   @  requires true;
6   @  ensures node.parent == old(node.parent);
7   @ }
8   @ ensures node.equals(old(node));
9 @*/
10 jpi void ExpEv(Exp node);

11 /*@ requires left != null && right != null;
12   @ model_program {
13   @  proceed(node, left, right);
14   @  requires node.left!=null && node.right!=null;
15   @  ensures node.parent == old(node.parent);
16   @ }
17   @ ensures node.equals(old(node));
18 @*/
19 jpi void BinEv(Exp node, Exp left, Exp right)
20  extends ExpEv(node);

22 /*@ requires left != null && right != null;
23   @ model_program {
24   @  proceed(node, left, right);
25   @  requires node.left!=null && node.right!=null;
26   @  ensures node.parent == old(node.parent);
27   @ }
28   @ ensures node.equals(old(node));
29 @*/
30 jpi void AndEv(Exp node, Exp left, Exp right)
31  extends BinEv(node, left, right);
32 /* subject */
33 class ASTVisitor exhibits AndEv,.. {
34  void visit(AndExp e) {
35   exhibit AndEv(e, e.left, e.right) {
36    e.left.accept(this);
37    e.right.accept(this);
38   };
39  } ..
40 }
41 /* observers */
42 aspect Evaluator {
43  Stack<BoolVal> valStack = ..
44  void around AndEv(Exp node, Exp left, Exp right){
45   proceed(node, left, right);
46   refining
47    requires node.left != null && node.right != null
48    ensures node.parent == old(node.parent){
49    .. // same as before
50   }
51  } ..
52 }
```

Figure 19. Join point interface `AndEv` and its translucid contract on lines 22–29.

Translucid contracts can be added to join point interfaces in a JML-like syntax, similar to join point types. Translucid contract of a join point interface appears right before its declaration. Figure 19 shows the translucid contract for the join point interface `AndEv`.

For the refining relation, in addition to syntactic adaptations of the refining rules, the rule (R-INVOKE) should be slightly modified to allow refinement of corresponding **`proceed`** statements with varying number of context variables in the translucid contracts of a join point interface and its supertype. A proceed statement in a translucid contract of a join point interface, refines a corresponding proceed statement in the translucid contract of its supertype, if the number of context variables of subtype's proceed is more than or equal to the number of context variables in supertype's proceed and types of context variables of the same names are the same. Join point interfaces do not support depth subtyping of context variables. For example, the proceed statement on line 13 of the translucid contract of `BinEv` refines its corresponding proceed statement on line 4 of the contract of `ExpEv`, i.e. $\mathtt{\textbf{proceed}}(node) \trianglelefteq \mathtt{\textbf{proceed}}(node, left, right)$. *Ptolemy*$_{\mathbb{S}}$'s specification inheritance, in Section B, could be adapted to join point interfaces, mostly through syntactic adaptations, to statically guarantee the refining between translucid contracts of a join point interface and its super join point interface.

For the non-decreasing relation, similar to join point types, precedence declarations of aspects could be statically checked to ensure that an aspect of a join point interface runs before aspects of its super join point interface or execution of aspects can be reordered dynamically at runtime to guarantee the non-decreasing relation.

For the conformance relation, similar to join point types, for each join point interface, all of its subjects and observers must conform to the translucid contract of the join point interface, i.e. structurally refine its model program and satisfy its preconditions and postconditions. Modified type checking rules of join point interfaces and runtime probes could enforce conformance.

D. Soundness of Reasoning

THEOREM D.1. *(Soundness of Ptolemy*$_{\mathbb{S}}$*'s Hoare logic) Ptolemy*$_{\mathbb{S}}$*'s Hoare logic, in Figure 8, is sound for conforming Ptolemy*$_{\mathbb{S}}$ *programs. In other words, any Hoare triple provable using Ptolemy*$_{\mathbb{S}}$*'s logic, i.e.* $\Gamma \vdash \{p\}\ e\ \{q\}$*, is a valid triple, i.e.* $\Gamma \models \{p\}\ e\ \{q\}$.

Proof: To prove the soundness of *Ptolemy*$_{\mathbb{S}}$'s Hoare logic, we must prove that each Hoare triple $\{p\}\ e\ \{q\}$ that is provable using *Ptolemy*$_{\mathbb{S}}$'s logic in Figure 8, i.e. $\Gamma \vdash \{p\}\ e\ \{q\}$, is a valid triple, i.e. $\Gamma \models \{p\}\ e\ \{q\}$, as defined in Section 4. The judgement $\Gamma \vdash \{p\}\ e\ \{q\}$ is valid, written as $\Gamma \models \{p\}\ e\ \{q\}$, if for every state σ that agrees with type environment Γ, if p is true in σ, i.e. $\sigma \models p$, and if the execution of e terminates in a state σ', then $\sigma' \models q$. This definition of validity is for partial correctness where termination is not guaranteed.

Previous work [24, 30, 31] proves the soundness of Hoare logic for object-oriented programs. Thus to prove the sounsness of *Ptolemy*$_{\mathbb{S}}$'s Hoare logic, it is sufficient to prove the soundness of *Ptolemy*$_{\mathbb{S}}$'s specific expressions [7], i.e. announce, invoke, refining and specificaiton expressions in the rules (V-ANNOUNCE), (V-INVOKE), (V-REFINING) and (V-SPEC) in *Ptolemy*$_{\mathbb{S}}$'s Hoare logic.

The proof is based on induction on the number of events, i.e. number of superevents of an event, in a subtyping hierarchy and the number of their observers and uses conformance, refining and non-decreasing relations. The induction goes over the number of superevents first and then number of observers.

D.1 Invoke Expression

For the Hoare logic rule (V-INVOKE) for an invoke expression, it should be proved that in an observer *ob* of an event *ev* if the Hoare triple $\{p\}$ **next.invoke**() $\{q\}$ is provable for its invoke expression, i.e. $\Gamma \vdash \{p\}$ **next.invoke**() $\{q\}$, then it is a validt Hoare triple, i.e. $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$. We assume an arbitrary chain of observers $\chi_0 \rightharpoonup ob \rightharpoonup \chi$ in which χ_0 contains observers in the chain before *ob* and χ is the remainder of the chain after *ob* with the event body at the end. The invoke expression in *ob* invokes the next observer in χ.

The first induction goes over the number of superevents of *ev* with base cases of zero and one superevent.

***No superevent for** ev* For the induction over the number of observers, we assume a base case with zero and one observer χ.

For the base case with zero observers, the invoke expression in *ob* causes the execution of the event body, say e', in χ. The subject conformance relation, in Definition 4.2, guarantees that the event body e' respects the precondition p' and postcondition q' of the top contract of *ev*, i.e. $\Gamma \models \{p'\}\ e'\ \{q'\}$. The top contract of *ev* is the same as the contract for *ev*, i.e. $p = p'$ and $q = q'$, because *ev* does not have any superevents. This in turn means that (a) $\Gamma \models \{p\}\ e'\ \{q\}$. Because the execution of **next.invoke**() in *ob* results in the execution of the event body, then in the judgement (a) the event body e' could be replaced with the invoke expression **next.invoke**() to arrive at the goal judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$.

For the base case with one observer ob_1 in χ, the invoke expression in *ob* causes the execution of the body e_1 of ob_1. The observer conformance relation, Definition 4.1, guarantees that the body e_1 of the observer ob_1 respects the precondition p and postcondition q of the contract of *ev*, i.e. (b) $\Gamma \models \{p\}\ e_1\ \{q\}$. And because the execution of **next.invoke**() in *ob* results in the execution, then in the judgement (b) the body e_1 of ob_1 could be replaced with the invoke expression to arrive at the goal judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$.

For the inductive case over the number of observers, we assume the induction hypothesis, i.e. the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds for the invoke expression in the observer *ob* with *n* observers in χ, and prove the judgement still holds for $n+1$ observers in χ. If the newly added observer is added, right after *ob*, to the beginning of χ, then the observer conformance relation guarantees that its body repsects the precondition and postcondition p and q of *ev* and the rest of the proof continutes as in the base case with one observer. If the newly added observer, is not added to the beginning of χ and is added somewhere down the chain χ, then using induction hypothesis the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds mainly because the hypothesis ensures the invoke expression causes the invocation of an observer which respects p and q.

The inductive proof of the invoke expression for the case in which there is no superevent for *ev* is similar to the proof of soundness of reasoning using translucid contracts in previous work [5], in the absence of event subtyping.

***One superevent** ev' **for** ev* For the induction over the number of observers in χ , we assume base cases with (1) zero observer for *ev* and ev', (2) one observer for *ev* and zero observer for ev', (3) zero observer for *ev* and one observer for ev' and (4) one observer for *ev* and one observer for ev'.

The proof for base case (1) is similar to the previous case with no superevent for *ev* and zero observers for *ev*. The subject conformance relation guarantees that the body e' of the event *ev* respects the precondition p' and postcondition q' of the top contract for *ev*, which is the contract of ev', i.e. (a) $\Gamma \models \{p'\}\ e'\ \{q'\}$. The refining relation guarantees that the contract of *ev* refines the contract of ev', i.e. $p \Rightarrow p'$ and $q \Rightarrow q'$. Using these implications among preconditions and postconditions, the judgement (a) and the standard rule (V-CONSEQ) one can arrive at the conclusion $\Gamma \models \{p\}\ e'\ \{q\}$ and replace e' with the invoke expression.

The proof for case (2) is similar to the previous case with no superevent for *ev* and one observer for *ev*.

For the case (3) the conformance relation guarantees that the body e'_1 of the only observer ob'_1 of ev' respects the contract of its event, i.e. (b) $\Gamma \models \{p'\}\ e'_1\ \{q'\}$. The refining relation guarantees that the contract of *ev* refines the contract of ev', i.e. $p \Rightarrow p'$ and $q \Rightarrow q'$. Using these implications, the judgement (b) and the rule (V-CONSEQ) one can arrive at the goal conclusion $\Gamma \models \{p\}\ e'_1\ \{q\}$ and then replace e'_1 with the invoke expression.

For the base case (4), the ordering relation guarantees that the only observer ob_1 of *ev* is before the only observer ob'_1 of ev' in the chain χ. The observer conformance relation guarantees that the body e_1 of ob_1 respects the precondition p and postcondition q of *ev*. The rest of the proof is similar to the base case with no superevent and one observer.

For the inductive case over the number of observers, we assume the induction hypothesis, i.e. the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds for the invoke expression in the observer *ob* of event *ev* with *n* observers of *ev* and its superevent ev' in χ, and prove the judgement holds for $n+1$ observers in χ. The newly added observer can be an observer of *ev* or ev'.

If the newly added observer is an observer of *ev* and it is added to the beginning of χ, the proof continutes similar to the inductive case for no superevent case in which a new observer is added to the beginning of χ. If the newly added observer of *ev* is not added to the beginning of χ, then the ordering relation guarantees that it is added before any observer of ev', then the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds mainly because the induction hypothesis ensures the invoke expression causes the invocation of an observer which respects p and q.

If the newly added observer is an observer of ev' then the ordering relation guarantees that it is added after any observer of *ev*, then the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds mainly because the induction hypothesis ensures the invoke expression causes the invocation of an observer which respects p and q.

k ***superevents for** ev* For induction over the number of superevents, we proved the base case with zero and one superevent for *ev*. For the inductive case we assume the induction hypothesis, i.e. the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds for the invoke expression in the observer *ob* of event *ev* with *n* observers of *ev* and its *k* superevents in χ, and prove the judgement holds for $k+1$ superevents with arbitrary number of observers for the newly added superevent.

If there are no observers in χ, i.e. $n = 0$, and the newly added superevent $ev^{(k)}$ has no observers too, then the proof is the same as the case with no superevent and no observers. If $ev^{(k)}$ has observers with $n = 0$ then the observer conformance relation guarantees that its first observers respect its precondition p^k and postcondition q^k and the refining relation guarantees that $p \Rightarrow p^k$ and $q \Rightarrow q^k$. Using these implications and the induction hypothesis we can arrive at the goal judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$, similar to the case for one superevent with no observer for the event and one observer for its superevent. If there are observers in χ, i.e. $n \neq 0$, then the ordering relation guarantees that observers of newly added superevent ev^k are added to the end of χ, and then the judgement $\Gamma \models \{p\}$ **next.invoke**() $\{q\}$ holds mainly because the induction hypothesis ensures the invoke expression causes the invocation of an observer which respects p and q.

D.2 Other Expressions

Announce expressions The proof for an announce expression is similar to the proof for the invoke expression, especially that the semantics of an announce expression is given in terms of invoke expression in the (ANNOUNCE) in Figure 13. Both announce and invoke expression cause execution of a chain of observers of an event and its superevents.

Refining and specification expressions For the refining expression in the rule (V-REFINING), the assumption of the rule that the body e of the refining expression satisfies its specification, i.e. $\Gamma \vdash \{p\}\ e\ \{q\}$ makes the conclusion valid. The validity of the rule (V-SPEC) is trivially true [29] and the rule (V-CONSEQ) is standard [24]. ∎

E. Soundness of Type System

(T-METHODDECL)

$$\frac{\begin{array}{c}(var:t)*, \texttt{this}:c \vdash e : \texttt{exp}\ t'' \\ t'' \preccurlyeq t' \qquad \texttt{class}\ c\ \texttt{extends}\ d\{..\} \qquad override(m, d, t* \to t')\end{array}}{\vdash t'\ m((t\ var)*)\ \{e\} : \text{ OK in } c}$$

(T-CALL)

$$\frac{\begin{array}{c}\Pi \vdash e : \texttt{exp}\ t \qquad t''\ m((t\ var)*)\ \{e'\} = CT(t, m) \\ \forall\, e_i \in e* \,\textbf{.}\ \Gamma, \Pi \vdash e_i : \texttt{exp}\ t_i' \qquad \forall\, t_i \in t*, t_i' \,\textbf{.}\ t_i' \preccurlyeq t_i\end{array}}{\Gamma, \Pi \vdash e.m(e*) : \texttt{exp}\ t''}$$

(T-NEW)

$$\frac{isClass(c)}{\Gamma, \Pi \vdash \texttt{new}\ c() : \texttt{exp}\ c}$$

(T-CAST)

$$\frac{isClass(c) \qquad \Gamma, \Pi \vdash e : \texttt{exp}\ t}{\Gamma, \Pi \vdash \texttt{cast}\ c\ e : \texttt{exp}\ c}$$

(T-GET)

$$\frac{\Gamma, \Pi \vdash e : \texttt{exp}\ c \qquad fieldsOf(c)(f) = t}{\Gamma, \Pi \vdash e.f : \texttt{exp}\ t}$$

(T-SET)

$$\frac{\Gamma, \Pi \vdash e : \texttt{exp}\ c \qquad fieldsOf(c)(f) = t \qquad \Gamma, \Pi \vdash e' : \texttt{exp}\ t' \qquad t' \preccurlyeq t}{\Gamma, \Pi \vdash e.f = e' : \texttt{exp}\ t'}$$

(T-DEFINE)

$$\frac{\begin{array}{c}\Gamma, \Pi \vdash e_1 : \texttt{exp}\ t_1 \\ \Gamma, \Pi, var : t \vdash e_2 : \texttt{exp}\ t_2 \qquad isType(t) \qquad t_1 \preccurlyeq t\end{array}}{\Gamma, \Pi \vdash t\ var = e_1; e_2 : \texttt{exp}\ t_2}$$

(T-IF)

$$\frac{\Gamma, \Pi \vdash e_1 : \texttt{exp}\ t \qquad \Gamma, \Pi \vdash e_2 : \texttt{exp}\ t \qquad \Gamma, \Pi \vdash ep : \texttt{exp}\ t}{\Gamma, \Pi \vdash \texttt{if}(ep)\{e_1\}\ \texttt{else}\ \{e_2\} : \texttt{exp}\ t}$$

(T-NULL)

$$\frac{isClass(c)}{\Gamma, \Pi \vdash \texttt{null} : \texttt{exp}\ c}$$

(T-VAR)

$$\frac{(var : t) \in \Gamma}{\Gamma, \Pi \vdash var : \texttt{var}\ t}$$

(T-LOC)

$$\frac{\Pi(loc) = t}{\Gamma, \Pi \vdash loc : \texttt{exp}\ t}$$

Figure 20. Standard *Ptolemy*$_{\mathbb{S}}$'s type checking rules [13].

Soundness proof of *Ptolemy*$_{\mathbb{S}}$'s type system follows standard progress and preservation arguments [36] using the refining and non-decreasing relations. Some details and definitions are adapted from previous work [5, 6, 12, 13]. Figure 17, Figure 18, Figure 13, Figure 14, Figure 20 and Figure 21 together show a complete list of *Ptolemy*$_{\mathbb{S}}$'s static and dynamic semantics rules.

E.1 Background Definitions and Lemmas

The following definitions are used in progress and preservation arguments of *Ptolemy*$_{\mathbb{S}}$'s soundness proof.

(NEW)

$$\frac{\begin{array}{c}loc \notin dom(S) \\ S' = S \uplus (loc \mapsto [c\,\textbf{.}\,\{f \mapsto \texttt{null} \mid f \in dom(fieldsOf(c))\}]) \\ \Pi' = \Pi \uplus (loc : c)\end{array}}{\langle \mathbb{E}[\texttt{new}\ c()], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[loc], S', \Pi', A\rangle}$$

(GET)

$$\frac{[c\,\textbf{.}\,F] = S(loc) \qquad v = F(f)}{\langle \mathbb{E}[loc.f], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[v], S, \Pi, A\rangle}$$

(SET)

$$\frac{[c\,\textbf{.}\,F] = S(loc) \qquad S' = S \uplus (loc \mapsto [c\,\textbf{.}\,F \uplus (f \mapsto v)])}{\langle \mathbb{E}[loc.f = v], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[v], S', \Pi, A\rangle}$$

(DEF)

$$\frac{e' = e[v/var]}{\langle \mathbb{E}[t\ var = v\,\textbf{;}\,e], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[e'], S, \Pi, A\rangle}$$

(CALL)

$$\frac{\begin{array}{c}[c.F] = S(loc) \qquad (c_2, t\ m((t\ var)*)\{e\} = methodBody(c, m) \\ e' = e[v*/var*, loc/\texttt{this}]\end{array}}{\langle \mathbb{E}[loc.m(v*)], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[e'], S, \Pi, A\rangle}$$

(CAST)

$$\frac{[c'\,\textbf{.}\,F] = S(loc) \qquad c' \preccurlyeq t}{\langle \mathbb{E}[\texttt{cast}\ t\ loc], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[loc], S, \Pi, A\rangle}$$

(NCAST)

$$\langle \mathbb{E}[\texttt{cast}\ c\ \texttt{null}], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[\texttt{null}], S, \Pi, A\rangle$$

(IFTRUE)

$$\frac{v \neq 0}{\langle \mathbb{E}[\texttt{if}(v)\{e_1\}\ \texttt{else}\{e_2\}], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[e_1], S, \Pi, A\rangle}$$

(IFFALSE)

$$\frac{v == 0}{\langle \mathbb{E}[\texttt{if}(v)\{e_1\}\ \texttt{else}\{e_2\}], S, \Pi, A\rangle \hookrightarrow \langle \mathbb{E}[e_2], S, \Pi, A\rangle}$$

(X-GET)

$$\langle \mathbb{E}[\texttt{null}.f], S, \Pi, A\rangle \hookrightarrow \langle \texttt{NPE}, S, \Pi, A\rangle$$

(X-SET)

$$\langle \mathbb{E}[\texttt{null}.f = v], S, \Pi, A\rangle \hookrightarrow \langle \texttt{NPE}, S, \Pi, A\rangle$$

Figure 21. *Ptolemy*$_{\mathbb{S}}$'s operational semantics for standard OO expressions, based on [13].

DEFINITION E.1. *(Location loc has type t in store S [13]) Location loc has type t in store S, written as $S(loc) : t$ where $t = \Pi(loc)$, if one of the following conditions hold:*

(I). type t is a class, i.e. isClass(t), and for some class name c with a set of fields F all the following holds:
  - *(a) $S(loc) = [c.F]$ and $\Pi(loc) = t$ and $c \preccurlyeq t$*
  - *(b) $dom(F) = dom(fieldsOf(c))$ and $rng(F) \subseteq (dom(S) \cup \{\texttt{null}\})$*
  - *(c) $\forall f \in dom(F)$ if $F(f) = loc'$ and $fieldsOf(c)(f) = u$ and $S(loc') = [u'.F']$ then $u' \preccurlyeq u$.*

*(II). type t is an event closure type, i.e. isThunkType(t), where $t =$ **thunk** ev for some event type ev with return type c, list of handlers H, environment ρ, expression e and class name c′ all the following holds:*
  - *(a) $S(loc) = \textbf{eClosure}(H, e, \rho)$*
  - *(b) $\Gamma, \Pi \vdash e : c'$ and $c' \preccurlyeq c$*
  - *(c) $\forall f \in dom(contextsOf(ev))$, either $\rho(f) = \texttt{null}$ or $\rho(f) = loc''$ where $S(loc'') = [c''.F']$ and $c'' \preccurlyeq contextsOf(ev)(f)$*

(d) $\forall h = \langle loc', m\rangle \in H$. $\Pi(loc') = c'''$ and $(c_2, c\ m(t_1\ var_1)) = methodBody(c''', m)$ then $t_1 = t$

DEFINITION E.2. *(Store S is consistent with store typing Π) Store S is consistent with store typing Π and typing context Γ, written as $\Gamma, \Pi \cong S$, if and only if all the following conditions hold:*

(a). $dom(S) = dom(\Pi)$
(b). $\forall loc \in dom(S), S(loc) : \Pi(loc)$, *i.e.* $S(loc)$ *has type* $\Pi(loc)$.

The following lemmas are used in progress and preservation arguments of *Ptolemy*$_{\mathbb{S}}$'s soundness proof. Proofs of these lemmas could be easily adapted from previous work on MiniMAO$_0$ [12] and thus are skipped.

LEMMA 1. *(Substitution) If $\Gamma, var_1 : t_1, .., var_n : t_n, \Pi \vdash e : t$ and $\forall i \in [1, n]$. $\Gamma, \Pi \vdash e_i : t'_i$ where $t'_i \preccurlyeq t_i$ then $\Gamma, \Pi \vdash e[var_1/e_1, .., var_n/e_n] : t'$ for some $t' \preccurlyeq t$.*

LEMMA 2. *(Environment contraction) If $\Gamma, a : t', \Pi \vdash e : t$ and a is not free in e, then $\Gamma, \Pi \vdash e : t$*

LEMMA 3. *(Environment extension) If $\Gamma, \Pi \vdash e : t$ and $a \in dom(\Gamma)$ then $\Gamma, a : t', \Pi \vdash e : t$*

LEMMA 4. *(Replacement) If $\Gamma, \Pi \vdash \mathbb{E}[e] : t$ and $\Gamma, \Pi \vdash e : t'$ and $\Gamma, \Pi \vdash e' : t'$ then $\Gamma, \Pi \vdash \mathbb{E}[e'] : t$.*

LEMMA 5. *(Replacement with subtyping) If $\Gamma, \Pi \vdash \mathbb{E}[e] : t$ and $\Gamma, \Pi \vdash e : u$ and $\Gamma, \Pi \vdash e' : u'$ such that $u' \preccurlyeq u$ then $\Gamma, \Pi \vdash \mathbb{E}[e'] : t'$ where $t' \preccurlyeq t$.*

E.2 Progress

THEOREM E.3. *(Progress) Let $\langle e, S, \Pi, A\rangle$ be a configuration with a well typed expression e, store S, store typing Π and active object map A, such that store S is consistent with store type Π, i.e. $\Gamma, \Pi \cong S$. If e has type t, i.e. $\Gamma, \Pi \vdash e : t$, then either*

- *$e = loc$ and $loc \in dom(S)$*
- *$e =$ **null**, or*
- *one of the following holds:*
  - $\langle e, S, \Pi, A\rangle \hookrightarrow \langle e', S', \Pi', A'\rangle$.
  - $\langle e, S, \Pi, A\rangle \hookrightarrow \langle x, S', \Pi', A'\rangle$ *and* $x \in \{$***NPE***, ***CCE***, ***TCE***$\}$

Proof sketch: Proof is by cases on evaluation of expression e:

1. $e = loc$. Since e is well-typed and using (T-LOC), $loc \in dom(\Pi)$. Using store consistency $\Gamma, \Pi \cong S$, $loc \in dom(S)$.
2. $e =$ **null**. The case is trivial.

Proof of cases for *Ptolemy*$_{\mathbb{S}}$'s announcement and handling of events, and registration and unregistration of observers are adapted from Ptolemy [13].

3. $e = \mathbb{E}[$**announce** $ev(v*)]$. Using well-typedness of e and (T-ANNOUNCE), event type ev is a declared event type in class table CT. (T-ANNOUNCE) ensures all the context variables of ev are passed to the announce expression with appropriate types which in turn allows (ANNOUNCE) to construct the event closure and take a step.
4. $e = \mathbb{E}[loc.$**invoke**$()]$. Using (T-INVOKE) and store consistency, $loc \in dom(S)$ and $\Pi(loc) =$ **thunk** ev which ensures loc is pointing to an event closure in the store for event ev. If the list of observer handlers H is not empty, then based on part (d) of Definition E.1 location loc', pointing to the first observer handler in the event closure, is well-typed and thus $loc' \in dom(S)$ which allows (INVOKE) to take an step. Otherwise, if H is empty, (INVOKEDONE) takes an step.
5. $e = \mathbb{E}[$**register**$(loc)]$. Using (T-REGISTER) and store consistency, (REGISTER) can take a step by adding a well-type location loc to the list of active objects $A[ev]$. The rule (T-BINDING) ensures that the event ev that observer instance loc is bound to, in the auxiliary function $eventsOf$, is a valid event type declared in class table CT.
6. $e = \mathbb{E}[$**unregister**$(loc)]$. Similar to previous case, using (T-UNREGISTER) and store consistency, (UNREGISTER) can take a step by removing a well-typed location loc from the list of active objects $A[ev]$.

Proof of cases for *Ptolemy*$_{\mathbb{S}}$'s checking of translucid contracts are:

7. $e = \mathbb{E}[$**refining requires** n **ensures** $q]$. The rule (T-REFINING) ensures precondition is well-typed which in turn allows (REFINING) to take an step and reduce to an **evalpost** expression, if the precondition holds, i.e. $n \neq 0$. Otherwise, (X-REFINING) takes a step.
8. $e = \mathbb{E}[$**evalpost** $n\ q]$. The rule (T-REFINING) ensures well-typedness of its postcondition q and body e, which in turn allows (EVALPOST) to take an evaluation step, if its postcondition holds, i.e. $n \neq 0$. In case the postcondition is violated, (X-EVALPOST) takes a step.

The following cases takes a step into exceptional terminal states and thus are trivial.

9. $e = \mathbb{E}[$**register**$($**null**$)]$, $e = \mathbb{E}[$**unregister**$($**null**$)]$, $e = \mathbb{E}[$**null**$.m(e*)]$, $e = \mathbb{E}[$**null**$.f]$, $e = \mathbb{E}[$**null**$.f = v]$, $e = \mathbb{E}[$**cast** c **null**$]$.

The following cases for standard object-oriented expressions either are trivial or could be easily adapted from MiniMAO$_0$ [12].

10. $e = \mathbb{E}[loc.f]$, $e = \mathbb{E}[loc.f = v]$, $e = \mathbb{E}[$**cast** $t\ loc]$, $e = \mathbb{E}[loc.m(v*)]$.
11. $e = \mathbb{E}[t\ \ var = v; e]$, $e = \mathbb{E}[$**if**$(v)\{e_1\}$ **else**$\{e_2\}]$, $e = \mathbb{E}[$**new** $c()]$ are trivial. ■

E.3 Preservation

THEOREM E.4. *(Preservation) Let e be an expression, S a store, Π a store typing and A a map of active objects where store S is consistent with store typing Π, i.e. $\Gamma, \Pi \cong S$. If $\Gamma, \Pi \vdash e : t$ and $\langle e, S, \Pi, A\rangle \hookrightarrow \langle e', S', \Pi', A'\rangle$ then $\Gamma | \Pi' \cong S'$ and there exists a type t' such that $t' \preccurlyeq t$ and $\Gamma | \Pi' \vdash e' : t'$.*

In the above definition Π' is the store typing built and maintained in *Ptolemy*$_{\mathbb{S}}$'s dynamic semantic rules.

Proof sketch: The proof is by cases on the evaluation relation $\hookrightarrow$:

Proofs for expressions which announce and handle events and (un)register observers are adapted from Ptolemy [6, 13].

1. (ANNOUNCE).
   $e = \mathbb{E}[$**announce** $ev\ (v*)\ \{e\}]$ and $e' = \mathbb{E}[loc.$**invoke**$()]$, where $(c$ **event** ev **extends** $ev'\{(t\ var)*\ contract_{ev}\}) \in CT$, $loc \notin dom(S)$, $H = handlersOf(ev)$, $\rho = \{var_i \mapsto v_i \mid var_i \in var* \wedge v_i \in v*\}$, $S' = S \uplus (loc \mapsto$ **eClosure**$(H, e, \rho))$, and $\Pi' = \Pi \uplus (loc :$ **thunk** $ev)$.

   To show the store consistency $\Gamma | \Pi' \cong S'$, part (a) of Definition E.2 holds since (ANNOUNCE) adds a fresh location loc to domains of both store S and store typing Π. Part (b) of store consistency definition holds for all locations $loc' \neq loc$, according to $\Gamma, \Pi \cong S$. To show that part (b) holds for loc, we have to show that part (II) of Definition E.1 holds for loc.

Part (a), of part (II) of Definition E.1 holds, since $S'(loc) = \texttt{eClosure}(H, e, \rho)$ and $\Pi'(loc) = \texttt{thunk}\ ev$. Part (b) holds since using (T-ANNOUNCE), the fact that because of the refining relation the return type of the event body e is the same as the top event of ev and considering that the return type c of ev is the supertype of the return type of its top event, if $\Gamma, \Pi \vdash e : c'$ then $c' \preccurlyeq c$. For part (c) for all $f \in dom(contextsOf(ev))$, $\rho(f) = \texttt{null}$ or $\rho(f) = loc''$. Part (c) holds trivially if $\rho(f) = \texttt{null}$. Otherwise if $\rho(f) = loc''$ according to store consistency $\Gamma, \Pi \cong S$, $loc'' \in dom(S)$. If $[c''.F] = S(loc'')$ then $\Gamma, \Pi \vdash loc'' : c''$ and (T-ANNOUNCE) ensures $c'' \preccurlyeq contextsOf(ev)(f)$. Then sing Lemma 3 we have $\Gamma|\Pi' \vdash loc'' : c''$ where $c'' \preccurlyeq contextsOf(ev)(f)$.

Now we show $\mathbb{E}[loc.\texttt{invoke}()] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash \texttt{announce}\ ev(v*)\{e\} : t$. Using (T-ANNOUNCE), $t\ \texttt{event}\ ev\ \texttt{extends}\ ev'\{..\} \in CT$ and using the relation between return types of the event body and the return type of events in its hierarchy for the refining relation, if $\Gamma, \Pi \vdash e : u$ then $u \preccurlyeq t$. Let $\Gamma, \Pi \vdash loc.\texttt{invoke}() : t'$. Using (T-INVOKE), $\Pi(loc) = \texttt{thunk}\ ev$ where $S(loc) = \texttt{eClosure}(H, e, \rho)$ such that $u \preccurlyeq t'$. Thus we have $u \preccurlyeq t$ and $u \preccurlyeq t'$ which means $t = t'$. Since subtyping relation $\preccurlyeq$ is reflexive, $t' \preccurlyeq t$.

2. (INVOKEDONE). $e = [loc.\texttt{invoke}()]$ and $e' = \mathbb{E}[e'']$, where $\texttt{eClosure}(\bullet, e'', \rho) = S(loc)$.

   Store consistency is trivial since neither store nor store typing changes.

   Now we show $\Gamma, \Pi \vdash \mathbb{E}[e''] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash e'' : u'$ and $\Gamma, \Pi \vdash loc.\texttt{invoke}() : u$. Using (T-INVOKE), $\Gamma, \Pi \vdash loc : \texttt{thunk}\ ev$ for some ev with return type u. Using store consistency and Definition E.1 part (II) item (b) and assumption $\texttt{eClosure}(\bullet, e'', \rho) = S(loc)$, we have $u' \preccurlyeq u$. Finally using Lemma 4, $t' \preccurlyeq t$.

3. (INVOKE).
   $e = \mathbb{E}[loc.\texttt{invoke}()]$ and $e' = \mathbb{E}[e_1[loc_1/var_1, loc'/\texttt{this}]]$, where $\texttt{eClosure}(\langle loc', m\rangle + H, e'', \rho) = S(loc)$, $[c\,.F'] = S(loc')$, $(c_2, t\ m(t_1\ var_1)\{e_1\}) = methodBody(c, m)$, $loc_1 \notin dom(S)$, $S' = S \uplus (loc_1 \mapsto \texttt{eClosure}(H, e'', \rho))$, and $\Pi' = \Pi \uplus (loc_1 : \Pi(loc))$.

   To show store consistency, $\Gamma|\Pi' \cong S'$, part (a) of Definition E.2 holds since (INVOKE) adds a fresh location loc_1 to the domain of both store S and store typing Π. Part (b) of store consistency definition holds for all locations $loc \neq loc_1$, using $\Gamma, \Pi \cong S$. To show that part (b) holds for loc_1 too, we have to show part (II) of Definition E.1 holds for loc_1. Part (a), of part (II) of Definition E.1 holds, since $S'(loc_1) = \texttt{eClosure}(H, e'', \rho)$ and $\Pi'(loc_1) = \Pi(loc)$. Using (T-INVOKE), $\Pi(loc_1)$ is an event closure thunk type $\texttt{thunk}\ ev$ for some event ev with return type c. Part (b) holds since using (T-ANNOUNCE), if $\Gamma, \Pi \vdash e'' : c'$ then $c' \preccurlyeq c$. For part (c) for all $f \in dom(contextsOf(ev))$, $\rho(f) = \texttt{null}$ of $\rho(f) = loc''$. Part (c) holds trivially if $\rho(f) = \texttt{null}$. Otherwise if $\rho(f) = loc''$ according to store consistency $\Gamma, \Pi \cong S$, $loc'' \in S$. If $[c''.F] = S(loc'')$ then $\Gamma, \Pi \vdash loc'' : c''$ and (T-ANNOUNCE) ensures $c'' \preccurlyeq contextsOf(ev)(f)$. Using Lemma 3 we have $\Gamma|\Pi' \vdash loc'' : c''$ where $c'' \preccurlyeq contextsOf(ev)(f)$.

   Now we show that $\mathbb{E}[e_1[loc_1/var_1, loc'/\texttt{this}]] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash loc.\texttt{invoke}() : u$ and $e_1 : u'$, which also hold in $\Gamma|\Pi'$, using Lemma 3. Using (T-INVOKE), $\Gamma|\Pi' \vdash loc : \texttt{thunk}\ ev$ for some ev with return type u. Location loc' in the event closure $\texttt{eClosure}(\langle loc', m\rangle + H, e'', \rho) = S'(loc)$ points to the class which contains the next handler method m to be run by the invoke expression. Expression e_1 is the body of m where using (T-BINDING) and (T-SUBEVENT), $u' \preccurlyeq u$. Using Lemma 1 $\Gamma|\Pi' \vdash e_1[loc_1/var_1, loc'/\texttt{this}] : u''$ such that $u'' \preccurlyeq u'$. Since $u' \preccurlyeq u$ and $u'' \preccurlyeq u'$, then $u'' \preccurlyeq u$. Using Lemma 4, $t' \preccurlyeq t$.

4. (ECGET). $e = \mathbb{E}[loc.f]$, $e' = \mathbb{E}[v]$ where $\texttt{eClosure}(H, e'', \rho) = S(loc)$ and $v = \rho(f)$.

   Showing store consistency is trivial.

   Now we show $\Gamma, \Pi \vdash \mathbb{E}[v] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash loc.f : u$ and $\Gamma, \Pi \vdash v : u'$. Using store consistency and part(c) of Definition E.1 part (II), $u' \preccurlyeq u$. And using Lemma 5, $t' \preccurlyeq t$.

5. (REGISTER). $e = \mathbb{E}[\texttt{register}(loc)]$, and $e' = \mathbb{E}[loc]$.

   Store consistency is trivial.

   Now we show $\Gamma, \Pi \vdash \mathbb{E}[loc] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash \texttt{register}(loc) : u$ and $\Gamma, \Pi \vdash loc : u'$. Using (T-REGISTER), $u' = u$. Using Lemma 5 we have $\Gamma, \Pi \vdash \mathbb{E}[loc] : t'$ for some $t' \preccurlyeq t$. Note that subtyping relation $\preccurlyeq$ is a reflexive transitive relation.

6. (UNREGISTER). $e = \mathbb{E}[\texttt{unregister}(loc)]$, and $e' = \mathbb{E}[loc]$. Similar to the case for (REGISTER).

Proofs for expressions that check translucid contracts are:

7. (REFINING). $e = \mathbb{E}[\texttt{refining requires}\ n\ \texttt{ensures}\ q\ \{e\}]$, $e' = \mathbb{E}[\texttt{evalpost}\ e\ q]$ where $n \neq 0$.

   Store consistency is trivial again.

   Now we show $\Gamma, \Pi \vdash \mathbb{E}[\texttt{evalpost}\ e\ q] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash [\texttt{refining requires}\ n\ \texttt{ensures}\ q\ \{e\}] : u$. Using (T-REFINNING), $\Gamma, \Pi \vdash e : u$. Using (T-EVALPOST) $\Gamma, \Pi \vdash \texttt{evalpost}\ e\ q : u$. Using Lemma 4 and reflexivity of subtyping relation we have $t' \preccurlyeq t$.

8. (EVALPOST). $e = \mathbb{E}[\texttt{evalpost}\ v\ n]$, $e' = \mathbb{E}[v]$ where $n \neq 0$.

   Store consistency is trivial since neither store nor store typing changes.

   Now we show $\Gamma, \Pi \vdash \mathbb{E}[v] : t'$ for some $t' \preccurlyeq t$. Let $\Gamma, \Pi \vdash v : u$. Using (T-EVALPOST), $\Gamma, \Pi \vdash \texttt{evalpost}\ v\ n : u$. Using Lemma 1 and reflexivity of subtyping relation we have $t' \preccurlyeq t$.

Proof for expressions that throw exceptions are the following.

9. (X-REFINING). $e = \mathbb{E}[\texttt{refining requires}\ n\ \texttt{ensures}\ q\ \{e\}]$, $e' = \texttt{TCE}$ where $n == 0$.

   Here e is reduced to an terminal condition $\texttt{TCE}$ which is not applicable to subject reduction theorem [12].

10. (X-SET), (X-GET), (X-CALL), (X-CAST), (X-REGISTER), (X-UNREGISTER), (X-EVALPOST). The same argument above, for (X-REFINING), applies to these rules too.

Proofs for standard object-oriented (OO) expressions are as the following:

11. Proofs for standard OO expressions as in rules (NEW), (SET), (GET), (CAST), (NCAST) and (CALL) could be easily constructed by adapting MiniMAO$_0$ [12] proofs for the same rules. ∎