Separating Obligations of Subjects and Handlers for More Flexible Event Type Verification

José Sánchez, Gary T. Leavens

Separating Obligations of Subjects and Handlers for More Flexible Event Type Verification

José Sánchez and Gary T. Leavens

University of Central Florida, Dept. of EECS, Orlando, FL 32816, USA
{sanchez, leavens}@eecs.ucf.edu

Abstract. Implicit invocation languages, like aspect-oriented languages, automate the Observer pattern, which decouples subjects (base code) from handlers (advice), and then compound them together in the final system. For such languages, event types have been proposed as a way of further decoupling subjects from handlers. In Ptolemy, subjects explicitly announce events at certain program points, and pass the announced piece of code to the handlers for its eventual execution. This implies a mutual dependency between subjects and handlers that should be considered in verification; i.e., verification of subject code should consider the handlers and vice versa.
However, in Ptolemy the event type defines only one obligation that both the handlers and the announced piece of code must satisfy. This limits the flexibility and completeness of verification in Ptolemy. That is, some correct programs cannot be verified due to specification mismatches between the announced code and the handlers' code. For example, when the announced code does not satisfy the specification of the entire event and handlers must make up the difference, or when the announced code has no effect, imposing a monotonic behavior on the handlers.
In this paper we propose an extension to the specification features of Ptolemy that explicitly separates the specification of the handlers from the specification of the announced code. This makes verification in our new language PtolemyRely more flexible and more complete, while preserving modularity.

Keywords: Event type, specification, verification, Ptolemy language

1 Introduction

Event types [12], and other similar approaches like XPIs [16], AAI [10], Open Modules[1, 11], IIIA with Join Point Types [15] and Joint Point Interfaces (JPI) [8, 5, 4], have been proposed as a way to further decouple subjects from handlers in implicit invocation and aspect-oriented languages. The verification systems for such languages should, as usual, strive to be as complete as possible while staying sound. In this work we propose some enhancements to the Ptolemy language and its specification and verification system for making it more complete while keeping it sound.

1.1 Completeness as a Measure of Usefulness

We work in the framework of a partial-correctness Hoare logic [7]. A judgement of the form $\Gamma \vdash \{P\}S\{Q\}$ means that the Hoare-triple $\{P\}S\{Q\}$ is provable using the type

environment Γ. The judgement $\Gamma \vdash \{P\}S\{Q\}$ is *valid* iff for every state σ that agrees with the type environment Γ, if P is true in σ (written $\sigma \models P$) and if the execution of S terminates in a state σ', then $\sigma' \models Q$. Such a logic is *sound* if whenever a judgment $\Gamma \vdash \{P\}S\{Q\}$ is provable, then it is valid. Conversely, such a logic is *complete* if whenever such a judgment is valid, then it is provable in the logic.

To compare two logics, one can ask if both are sound, and if so one can compare how complete they are. Logic A is *strictly more complete than* logic B if there is some valid judgment that is provable in A but not in B, but every judgment that is provable in B is provable in A. Given that both logics are sound, then a more complete logic is potentially more useful for users, as they will be able to prove more programs correct.

1.2 A Brief on Ptolemy Language

Ptolemy's [12] event type concept decouples subjects (*base code*), which explicitly announce *events*, from the *handlers* that process these events. The event type establishes the contract every handler must satisfy. In this way the base (or announcing) code can be modularly reasoned about using the contract, instead of using each handler's code. The contract not only defines the precondition and postcondition every handler method should satisfy, but also the abstract algorithm they must refine, called a *translucid* contract [3]. In the body of a translucid contract, *specification expressions* can abstract away details of particular implementation expressions, by only specifying their effects. *Invoke expressions* in the contract's body show where a handler triggers the execution of the next handler in the *execution chain* (until eventually reaching the originally announced code that stands at the end). In the base code, *announce expressions* are used to explicitly announce occurrences of events, starting the *execution chain* and passing the *announced code* to it. All this is schematized in Figure 1.

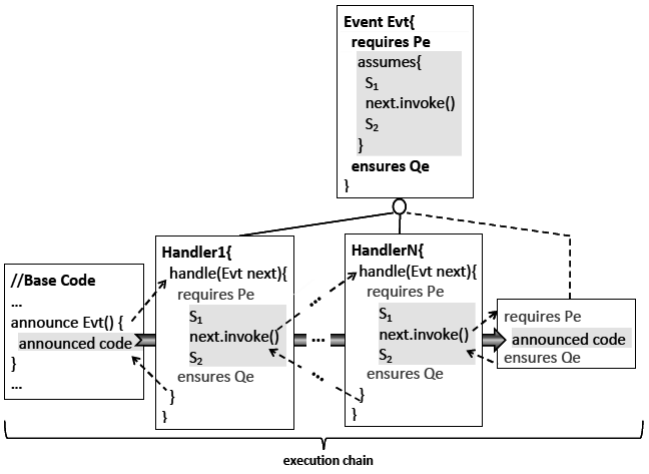

Fig. 1. Event, handlers and announced code

The *invoke expressions* in the contract make visible the control effects of the handlers. Active handlers are registered as such using *register expressions* and handlers are bound to the corresponding event by *when expressions*.

In Ptolemy, every handler method must be verified to satisfy the contract's pre- and postconditions and also to structurally refine (*see section 2*) the translucid contract's body, providing conforming implementations for every specification expression.[1] The announced code is also verified to satisfy the same contract's pre- and postconditions.

1.3 The Billing Example

The *billing* system example in Figure 2 illustrates the basic concepts of Ptolemy and motivates our proposed extension. In this system, each bill includes the amount (a) to be paid and the extra charges (c) like taxes. When the base code totals a bill, adding the charges to the principal amount (line 7), the corresponding event is announced (lines 6-8). This gives registered handlers (maybe $PaymentHandler$ or $ShippingHandler$) the chance to do some adjustments, like adding some extra charges. In this case we register just one handler at random (line 5) to emphasize the fact that the reasoning is based on the event definition, instead of the particular implementation of any specific handler. The *TotalingEvent* definition specifies the behavior and abstract algorithm of every admissible handler. The **`requires`** (line 14) and **`ensures`** (line 21)[2] clauses specifies the behavior: every handler requires (line 14) that the existing charges are not negative and ensures (line 21) that the resulting amount of the bill is greater than or equal to the sum of the original amount plus the original charges. The excess, if any, is due to the extra charges added by the handlers. The *translucid contract* (lines 16-19, inside **`assumes`**`{...}`) forces the handlers to make the charges greater than or equal to their current value, but allows charges to be added by each handler in any consistent way. The specification expression (lines 17-18) must be refined by each conforming handler, with code that satisfies the stated pre-post conditions. Also any **`invoke`** expression must be made explicit in the translucid contract (as on line 19). This allows modular verification of control effects, using the specification of the announced event.

This example is verified by Ptolemy's proof system. Both handlers refine the event's translucid contract. The specification expression in this contract (lines 17-18) is refined by $PaymentHandler$ by increasing the charges ($c' = c + 1$, line 27), and by $ShippingHandler$ by leaving the charges the same ($c' = c + 0$, line 38). Considering the above and the effect of the **`invoke`** expression, it can be seen that both handlers satisfy the event specification ($a' \geq a + c$, line 21), and so both are proven valid. The announced code ($a' = a + c$, line 7) also satisfies the event specification ($a' \geq a + c$, line 21), as required by Ptolemy's proof system, so the complete **`announce`** expression (lines 6-8) is proven valid. With the handlers and the **`announce`** expressions proven valid, the entire program is proven valid in Ptolemy.

[1] Ptolemy is an expression language.

[2] When summarizing assertions, we adopt the Z [14] convention of denoting the new value of a variable with a prime (like a'), and use unprimed variables to stand for their pre-state values.

```
1  public class Base {
2    public void run(){
3      Bill bill=new Bill(100,8);
4      Bill old=new Bill(bill.a(),bill.c());
5      registerHandler();// Randomly register one handler
6      announce TotalingEvent(bill) {  // event Q_e : a' ≥ a + c
7         bill.setA(bill.a()+bill.c());// code Q_s : a' = a + c
8      }
9      //assert bill.a()>old.a()+old.c(); //a' > a + c ??
10 } }
11
12 public void event TotalingEvent { // handlers: a' ≥ a + c
13   Bill bill;
14   requires (bill.c()>=0) // P_e: c ≥ 0
15   assumes{
16     // specification expr.: requires c ≥ 0 ensures c' ≥ c
17     requires (next.bill().c()>=0)
18       ensures (next.bill().c()>=old(next.bill().c()));
19     next.invoke(); // control flow: proceed with next handler
20   }
21   ensures (bill.a()>=old(bill.a())+old(bill.c())) //Q_e: a' ≥ a + c
22 }
23 public class PaymentHandler { // Payment Processing Fee Handler
24   public void handleTotaling(TotalingEvent next)throws Throwable{
25     refining requires (next.bill().c()>=0)
26              ensures (next.bill().c()>=old(next.bill().c())){
27       next.bill().setC(next.bill().c()+1);  // c' = c + 1
28     }
29     next.invoke();
30   }
31   when TotalingEvent do handleTotaling;
32   public PaymentHandler(){ register(this); }
33 }
34 public class ShippingHandler { // Shipping Fee Handler
35   public void handleTotaling(TotalingEvent next)throws Throwable{
36     refining requires (next.bill().c()>=0)
37              ensures (next.bill().c()>=old(next.bill().c())){
38       next.bill().setC(next.bill().c()+0); //c' = c + 0 NO FEE NOW
39     }
40     next.invoke();
41   }
42   when TotalingEvent do handleTotaling;
43   public ShippingHandler(){ register(this); }
44 }
```

Fig. 2. Billing example in Ptolemy

1.4 Completeness Issues: Enforcing the Billing "Increasing" Property

Now we consider a variation on the *billing* system. A new "business rule" requires us to enforce the "increasing" property: that all the handlers for *TotalingEvent* must strictly increase the total amount, by adding to the charges. Currently $PaymentHandler$ satisfies this condition (line 27) but $ShippingHandler$ does not (line 38). If this property were met, **the assertion on line 9 could be proven true**, since no matter which handler were registered (line 5) the charges would have been incremented.

We have to guarantee that any handler H bound to the event *TotalingEvent*, satisfies the required property, while keeping the program valid.[3] For doing that we can adjust the event specification and the handlers.

Definition 1. *An implementation of the* ***billing*** *program* ***satisfies the "increasing" property*** *if for each binding clause of the form* ***when*** $TotalingEvent$ ***do*** m *appearing in a class* C*: if* $H = bodyOf(C, m)$ *then* $\Gamma' \models \{c \geq 0\}H\{a' > a + c\}$.

The current *TotalingEvent* specification does not guarantee the above property, as its postcondition $(a' \geq a + c)$ does not imply $(a' > a + c)$. The way for the $billing$ system to satisfy this property is by having an event postcondition Q_e such that $Q_e \Rightarrow (a' > a+c)$. However in Ptolemy this Q_e must be such that $(a' = a+c) \Rightarrow Q_e$, to meet the requirement of Ptolemy's proof system that the announced code (line 7) satisfies the event specification. The fact that these two implications result in a contradiction shows that the above property cannot be proved in Ptolemy. This shows the incompleteness of Ptolemy's proof system, that is incapable of *modularly* proving the assertion in line 9.

In section 3 we propose an extension to Ptolemy that makes verification more flexible and complete, and in particular able to enforce the "increasing" property and verify the aforementioned assertion. First, we explain Ptolemy verification in more detail.

2 Verification in Ptolemy

In Ptolemy, event types state the obligations that handlers should satisfy. In the general case that was presented in Figure 1, the event *Evt*'s declaration specifies the precondition (P_e) and postcondition (Q_e) that handlers should conform to, and also the translucid contract (assumes clause) that they should refine.

Verification in Ptolemy is straightforward [3]. Every handler body H for an event and every piece of announced code S for that event must satisfy the same pre-post obligations [3, Figure 11], declared in the event's `requires` and `ensures` clauses. Besides that, the handlers must also refine the event's translucid contract. This is expressed in the requirement that a program is conformal, meaning that each handler conforms to the corresponding event declaration's specification.

Definition 2. *A Ptolemy program Prog* is conformal *if and only if for each declaration of an event type,* Evt*, in Prog, and for each binding clause of the form* ***when*** Evt ***do*** m *appearing in a class* C *of Prog: if* $(P_e, A, Q_e) = ptolemySpec(Evt)$ *and* $H = bodyOf(C, m)$*, then there is some type environment* Γ' *such that* $\Gamma'($***next***$) = closure\ Evt$*,* $\Gamma' \vdash A \sqsubseteq H$ *and* $\Gamma' \models \{P_e\}H\{Q_e\}$.

[3] The auxiliary function $bodyOf(C, m)$ returns the body of method m in class C.

In the above, the formula P_e is the event's precondition, Q_e is its postcondition, and A is the body of the **`assumes`** clause (the "translucid contract" [3]), which in our notation is written $(P_e, A, Q_e) = ptolemySpec(Evt)$. Similarly, $bodyOf(C, m)$ returns the code that is the body of method m in class C.[4] The structural refinement relation $\sqsubseteq$ is explained below. Furthermore, we say that a Hoare-triple $\{P\}S\{Q\}$ is *valid*, written $\Gamma \models \{P\}S\{Q\}$, if in every state (typable by Γ) such that P holds, whenever S terminates normally, then Q holds in the resulting state.

In Ptolemy, the verification of handlers is done modularly and separately from the announcements. The body of each handler must structurally refine the translucid contract from the event specification. A handler body, H, *structurally refines* a translucid contract A, written $A \sqsubseteq H$, if one can match each expression in H to an expression in A [13]. The matching of most expressions are exact (only the same expression matches) with the exception of specification expressions of the form **`requires`** P **`ensures`** Q, which can occur in A and must each be matched by expressions in H of the form **`refining requires`** P **`ensures`** Q **`{`** S **`}`**, where S is the code implementing the specification expression. In Ptolemy structural refinement is checked by the type checking phase of the compiler [3].

To summarize, according to the work on translucid contracts for Ptolemy [3], the way that one proves that a program is conformal is by proving, for each handler body H for an event Evt such that $(P_e, A, Q_e) = ptolemySpec(Evt)$: $\Gamma' \vdash A \sqsubseteq H$ and $\Gamma' \vdash \{P_e\}H\{Q_e\}$. In order to guarantee soundness, the body of each **`refining`** expression must satisfy the given specification, as in the (REFINING) rule of Figure 3.

$$\text{(SPECIFICATION-EXPR)}\quad \frac{}{\Gamma \vdash \{P\}\textbf{requires}\ P\ \textbf{ensures}\ Q\{Q\}}$$

$$\text{(REFINING)}\quad \frac{\Gamma \vdash \{P\}S\{Q\}}{\Gamma \vdash \{P\}(\textbf{refining requires}\ P\ \textbf{ensures}\ Q\ \{\ S\ \})\{Q\}}$$

$$\text{(ANNOUNCE)}\quad \frac{(P_e, A, Q_e) = ptolemySpec(Evt), \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \quad \Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}{\Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}\ \textbf{announce}\ Evt(\boldsymbol{y})\ S\ \{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}$$

$$\text{(INVOKE)}\quad \frac{closure\ Evt = \Gamma(\textbf{next}), \qquad (P_e, A, Q_e) = ptolemySpec(Evt)}{\Gamma \vdash \{P_e\}\ \textbf{next.invoke()}\ \{Q_e\}}$$

Fig. 3. Hoare Logic axioms and inference rules for the interesting constructs of Ptolemy.

[4] These auxiliary functions query the program, which is treated as a fixed context.

For every **announce** expression in a valid program, the announced code S should satisfy the event specification (P_e, Q_e). Then, if the base code guarantees P_e before the **announce** expression it can assume Q_e holds afterwards. This constitutes Ptolemy's (ANNOUNCE) rule in Figure 3. In that rule $P_e[\boldsymbol{y}/\boldsymbol{x}]$ means P_e with the actual parameter variables y_i[5] simultaneously substituted for the free occurrences of the x_i, which are the event's formal parameters. Note that the body of the announcement, S, cannot use the event's formal parameters, but only has access to the original type environment, Γ. In the (ANNOUNCE) rule, there is no distinction made regarding the presence or absence of any registered handlers, because the same reasoning applies in either case.

An **invoke** expression in a handler is reasoned about in the same way. That is, the code executing the invoke expression must establish P_e and can assume Q_e afterwards. This is (INVOKE) rule in Figure 3. In this rule, the event's name is obtained from the type of **next**, and this gives access to the specification (P_e, A, Q_e) of that event.

A Hoare logic is *sound* if whenever $\Gamma \vdash \{P\}S\{Q\}$ is provable then every terminating execution of S starting from a state in which P holds ends in a state in which Q holds. Soundness for Ptolemy depends on the program being conformal.

Theorem 1. *Suppose that the Hoare logic for Ptolemy, without using the rules in Figure 3, is sound. Then for conformal programs, the whole logic, including the rules in Figure 3, is sound.*

We omit the proof (which goes by induction on the structure of the proof in the entire Hoare logic). However, the key argument is the same as that for greybox specifications, that structural refinement implies refinement [13].

Ptolemy's design makes both handlers and the announced code have the same pre-post specifications (P_e, Q_e).[6] This design is convenient in some cases, but it limits Ptolemy's flexibility and completeness. For example, it is not possible to use Ptolemy's event type pre and postconditions to specify and verify the "increasing" property of our *billing* system (section 1.4), because the announced code achieves the postcondition $a' = a + c$ and not the event's postcondition $a' > a + c$. However, this property could be considered correct with respect to a more flexible specification that gives different postconditions to the announced code and handlers, which is what we do below. This example shows that verification in Ptolemy is incomplete.

We have other similar examples that show incompleteness of Ptolemy's verification rules. The common theme, like in the *billing* example, is that the effect of the announced code does not match the effect of the handlers.

Another situation that shows Ptolemy's incompleteness occurs when the announced code has no effect (e.g., **skip**). As Ptolemy imposes the event pre-post obligations on the announced code, it requires that the triple $\{P_e\}$**skip**$\{Q_e\}$ holds, or, by Hoare logic, that $P_e \Rightarrow Q_e$. Since these same obligations are imposed on the handlers, thus they are limited to monotonic behaviors; i.e. ones that preserve the precondition P_e. This is a symptom of incompleteness, because in a program where there must be registered

[5] We use variables in these rules to avoid problems with side effects in expressions, although Ptolemy allows general expressions to be passed as actual arguments to announcements.

[6] We use the convention of denoting by (P, Q) the pre- and postconditions of some code.

handlers, one would not be able to verify an event announcement in which the handlers achieve a postcondition Q_e that is not implied by the event's precondition (P_e).

In the next section we detail our proposed modification to solve these incompleteness issues and analyse its impact regarding modular reasoning.

3 Explicit Separate Specification

A solution to the incompleteness problems can be found by recognizing that there is a mutual dependency between base code, handlers and announced code, in the execution chain. The base code depends on the behavior of the activated handlers that are triggered by an **announce** expression. The handlers depend on the other activated handlers, and on the behavior of the announced code at the end of the chain (Figure 4).

The first change from Ptolemy in our *PtolemyRely* language consists in separating the specification for the handlers (P_e,Q_e) from the specification for the announced code (P_s,Q_s). As before, every handler H is reasoned about using the event requires-ensures specification (P_e,Q_e). But the announced code S is reasoned about using its own specification (P_s,Q_s). (Both cases are depicted in Figure 5). This new approach allows different specifications for the handlers and for the announced code, as in our *billing* example. This also allows announced code that has no effect to be verified without limiting, in any way, the handlers' specification.

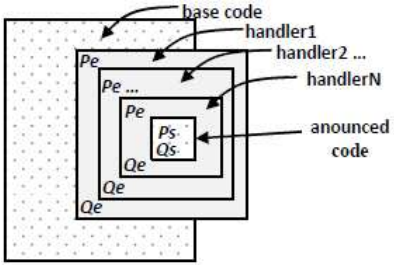

Fig. 4. Mutual dependencies between base code, handlers and announced code

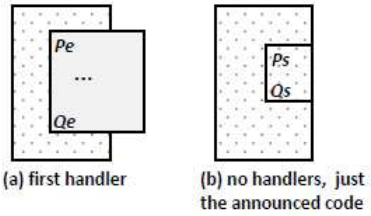

Fig. 5. Reasoning about the base code

In PtolemyRely, the second change is that the verification of both **announce** and **invoke** expressions is slightly modified. For **announce** expressions there are two situations, as shown in Figure 5. If there are registered handlers then the base code interacts with the first of them, which guarantees the event postcondition (Q_e). If there are no handlers then the announced code is executed, ensuring its postcondition (Q_s). This two cases are formalized by the rules (RANNOUNCEHAS) and (RANNOUNCENONE) in Figure 8.

invoke expressions are only valid inside the body of a handler, and thus should be analyzed in a context where there are registered handlers. Their effect, instead, depend on the nondeterministic position of the containing handler in the execution chain.

If there are other handlers left in the execution chain, the event specification (P_e, Q_e) is used, as all handlers satisfy it. If only the announced code is left, its specification (P_s, Q_s) should be used. However, for modular verification, the problem is that the event declaration, and consequently the handlers, do not know the announced code and thus do not know (P_s, Q_s). To avoid whole-program reasoning, we make a third change, in this case to Ptolemy's event type declarations. Now users also specify, in the event declaration, the pre-post obligations (P_r, Q_r) for any announced code. Putting this specification in the event type declaration in a new **`relies`** clause (see Figure 6) allows the handlers to be verified based on that specification, instead of the actual announced code's specification. It also allows one to avoid doing the verification that each handler satisfies the event pre-post specification one handler at time. Instead, that can be done in two separate steps: first, once and for all, verifying that the event's translucid contract satisfies the event's pre-post specification, and then verifying that each handler refines this translucid contract, which in turn guarantees every handler satisfies the event's specification.

To summarize, with our changes in PtolemyRely, the event type declares specifications for the handlers, (P_e, Q_e), and for the announced code, (P_r, Q_r). In the rest of this section, we give the formal details of our approach.

3.1 Syntax

For PtolemyRely, we change the syntax of Ptolemy event declarations by introducing a **`relies`** clause that establishes the specification for the announced code (P_r, Q_r). This is shown in the event syntax schema, Figure 6.

```
t event Evt {
  t1 f1;...; tn fn;
  relies requires Pr ensures Qr
  requires Pe
  assumes { ... next.invoke(); ...}
  ensures Qe
}
```

Fig. 6. Event syntax schema.

sp ::= ... | **`handlers`** (*c*)
contract ::= ... |
`relies requires` *sp* **`ensures`** *sp*
`requires` *sp*
`assumes` { *se* }
`ensures` *sp*

Fig. 7. Formal syntax changes.

We make two changes to the formal syntax of Ptolemy [3]. The first adds a predicate **`handlers`** that returns the number of handlers currently registered for its event argument. The second changes contract definitions, as shown in Figure 7. The nonterminal *c* stands for event names, *sp* stands for specification predicates, and *se* stands for specification expressions (the contract's body in this case).

3.2 Semantics

In PtolemyRely, as stated in the definition of conformance, we check for structural refinement of each handler to the translucid contract, and also check each handler to satisfy the event *requires-ensures* specification.

Definition 3. *A PtolemyRely program Prog* is conformal *if and only if for each declaration of an event type, Evt, in Prog, and for each binding clause of the form* **when** Evt **do** m *appearing in a class C of Prog: if $(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt)$ and $H = bodyOf(C, m)$, then there is some type environment Γ' such that $\Gamma'(\textbf{next}) = closure\ Evt$, $\Gamma' \vdash A \sqsubseteq H$, and $\Gamma' \models \{P_e\}H\{Q_e\}$.*

The function $eventSpec(Evt)$ returns the specification information from the event type's declaration. The returned 5-tuple consists of the relies clause contract (P_r, Q_r), and the translucid contract: pre and post-conditions (P_e, Q_e) and assumes body A.

The **announce** and **invoke** expressions are verified using the rules in Figure 8. For **announce** expressions there are two rules, depending on whether one can prove that there are registered handlers for the event. (RANNOUNCEHAS) applies when there are registered handlers. In this case the **announce** expression is reasoned about using the event's specification (P_e, Q_e). For this rule to be valid, the announced code, S, must satisfy the specification (P_r, Q_r) given in the event's type. (RANNOUNCENONE) applies when there are no registered handlers. In this case only the announced code is executed, and thus the relied on specification (P_r, Q_r) is used.

(RANNOUNCEHAS)

$$\frac{\begin{array}{c}(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt),\\ \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \qquad \Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}] \wedge \textbf{handlers}(Evt) > 0\} S \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}\end{array}}{\Gamma \vdash \{P_e[\boldsymbol{y}/\boldsymbol{x}]\}\ (\textbf{announce}\ Evt(\boldsymbol{y})\ S)\ \{Q_e[\boldsymbol{y}/\boldsymbol{x}]\}}$$

(RANNOUNCENONE)

$$\frac{\begin{array}{c}(P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt),\\ \boldsymbol{x} : \boldsymbol{T} = formals(Evt), \qquad \Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}] \wedge \textbf{handlers}(Evt) = 0\} S \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}\end{array}}{\Gamma \vdash \{P_r[\boldsymbol{y}/\boldsymbol{x}]\}\ (\textbf{announce}\ Evt(\boldsymbol{y})\ S)\ \{Q_r[\boldsymbol{y}/\boldsymbol{x}]\}}$$

(RINVOKE)

$$\frac{closure\ Evt = \Gamma(\texttt{next}), \qquad (P_r, Q_r, P_e, A, Q_e) = eventSpec(Evt)}{\Gamma \vdash \{P_e \wedge P_r\}\texttt{next.invoke()}\{Q_e \vee Q_r\}}$$

Fig. 8. Hoare Logic inference rules for those constructs of PtolemyRely that differ from Ptolemy.

The soundness theorem for PtolemyRely states that if a program is conformal, then all provable Hoare triples are valid.

Theorem 2 (Soundness). *Suppose that the Hoare logic for Ptolemy, without using the rules for* **invoke** *and* **announce**, *is sound. Then for conformal PtolemyRely programs, the whole logic, including the rules for those constructs in Figure 8, is sound.*

Proof: Let Γ, P, S and Q be given such that $\Gamma \vdash \{P\}S\{Q\}$ is provable using PtolemyRely's Hoare logic, including the rules in Figure 8. We prove that $\Gamma \models \{P\}S\{Q\}$ (i.e., that this Hoare triple is valid) by induction on the structure of the proof of that triple. In the base case, there are no uses of the rules in Figure 8, so validity follows by the hypothesis. For the inductive case, suppose that the proof has as its last step one of the rules in Figure 8. We assume inductively that all subsidiary proofs are valid. There are three cases. If the last step uses the (RANNOUNCENONE) rule, then the hypothesis that the announced code satisfies the specification (P_r, Q_r) makes the conclusion valid. If the last step uses the (RANNOUNCEHAS) rule, then the hypothesis that the program is conformal means that, by definition 3, $\Gamma' \models \{P_e\}H\{Q_e\}$ where (P_e, Q_e) is the specification of the handler's from the event type. This again makes the conclusion valid. If the last step uses the (RINVOKE) rule, then there are two sub-cases, and the proof is similar to that given for the previous cases, using the definition of "conformal." ∎

We note that proving that a program is conformal can be done in a simple way, by proving $\Gamma' \vdash \{P_e\}A\{Q_e\}$, where (P_e, Q_e) is the event's pre/post specification, and A is the translucid contract for the event, and then checking that each handler's body, H structurally refines the translucid contract ($\Gamma' \vdash A \sqsubseteq H$). After that, it follows that $\Gamma' \models \{P_e\}H\{Q_e\}$ using techniques from the work of Shaner *et al.* [13].

3.3 Billing Example Revisited (PtolemyRely)

In Figure 9 we show how our *billing* example could be written in PtolemyRely. Here we show how it can be verified using PtolemyRely's rules, how the "increasing" property can be specified and verified and how the assertion in line 9 is now proved.

Contrary to Ptolemy, PtolemyRely allows us to have different specifications for the handlers (P_e, Q_e) and for the announced code (P_s, Q_s). As mentioned before, the specification for the handlers, (P_e, Q_e), goes in the **`requires-ensures`** clauses of the event declaration, meanwhile the minimum specification for any announced code, (P_r, Q_r), goes in the new **`relies`** clause. The specification of the announced code S (line 7) is (P_s, Q_s), that corresponds to $(c \geq 0, a' = a + c)$. We take the expected behavior for the announced code (P_r, Q_r) (lines 13-14) to be the same as the actual behavior for the announced-code (P_s, Q_s). The specification for the handlers (P_e, Q_e) is declared in line 15 and line 22 as $(c \geq 0, a' > a + c)$.

In PtolemyRely we can prove our "increasing" property: that all handlers should strictly increase the total amount of the bill. If a handler H is verified, it means that it satisfies the (P_e, Q_e) specification. In this case: $\Gamma \vdash \{c \geq 0\}H\{a' > a + c\}$, which is exactly what the "increasing" property demands.

Since there are registered handlers (line 5) the (RANNOUNCEHAS) rule applies. It requires $\{P_r\}S\{Q_r\}$, which holds in the **`announce`** expression in lines 6-8. The postcondition in the consequent of this rule, Q_e, corresponds in this case to $a' > a + c$, this immediately proves the assertion in line 9. To reason about **`invoke`** expressions one should use the (RINVOKE) rule, that considers (P_e, Q_e) and (P_r, Q_r). In this case it corresponds to the following:

$$\Gamma \vdash \{(c \geq 0) \wedge (c \geq 0)\}\texttt{next.invoke()}\{(a' > a + c) \vee (a' = a + c)\}$$

and this is equivalent to $\Gamma \vdash \{c \geq 0\}\texttt{next.invoke()}\{a' \geq a + c\}$

```
1  public class Base {
2    public void run(){
3      Bill bill=new Bill(100,8);
4      Bill old=new Bill(bill.a(),bill.c());
5      registerHandler();// Randomly register one handler
6      announce TotalingEvent(bill) {  // event Q_e : a' ≥ a + c
7         bill.setA(bill.a()+bill.c());// code Q_s : a' = a + c
8      }
9      assert bill.a()>old.a()+old.c(); //a' > a + c
10 } }
11 public void event TotalingEvent { // handlers: a' > a + c
12   Bill bill;
13   relies requires bill.c()>=0 // announced code: P_r: c ≥ 0
14     ensures bill.a()==old(bill.a())+old(bill.c()) //Q_r:a' = a + c
15   requires (bill.c()>=0) // //handlers: P_e: c ≥ 0
16   assumes{
17     // specification expr.: requires c ≥ 0 ensures c' ≥ c
18     requires (next.bill().c()>=0)
19       ensures (next.bill().c()>=old(next.bill().c()));
20     next.invoke(); // control flow: proceed with next handler
21   }
22   ensures (bill.a()>old(bill.a())+old(bill.c())) //Q_e: a' > a + c
23 }
24 public class PaymentHandler { // Payment Processing Fee Handler
25   public void handleTotaling(TotalingEvent next)throws Throwable{
26     refining requires (next.bill().c()>=0)
27               ensures (next.bill().c()>=old(next.bill().c())){
28       next.bill().setC(next.bill().c()+1);  // c' = c + 1
29     }
30     next.invoke();
31   }
32   when TotalingEvent do handleTotaling;
33   public PaymentHandler(){ register(this); } }
34 public class ShippingHandler { // Shipping Fee Handler
35   public void handleTotaling(TotalingEvent next)throws Throwable{
36     refining requires (next.bill().c()>=0)
37               ensures (next.bill().c()>=old(next.bill().c())){
38       next.bill().setC(next.bill().c()+5); //c' = c + 5
39     }
40     next.invoke();
41   }
42   when TotalingEvent do handleTotaling;
43   public ShippingHandler(){ register(this); } }
```

Fig. 9. Billing example revisited (PtolemyRely)

In this *revisited* version we adjusted *ShippingHandler* to meet the "increasing" property (line 38). Both handlers refine the translucid contract, providing code (line 28

and 38) that correctly refines the specification expression in the contract (lines 18-19). Also both, *PaymentHandler* and *ShippingHandler*, satisfy the handlers specification $(c \geq 0, a' > a + c)$. This can be shown as follows. Both increment the charges, $c' > c$, (line 28 and line 38) and then invoke the next handler. Considering this increment, and the indicated postcondition of the **`invoke`** expression, we have $(c' > c) \wedge (a' \geq a + c')$, and from that we get $(a' > a + c)$, that shows that both handlers satisfy the specification.

We have showed that the whole program is verified (announce expression and handlers), that the "increasing" property can also be verified and that the assertion in line 9 can be proved in PtolemyRely.

3.4 Extension of Ptolemy

Our new approach extends Ptolemy's, as stated in the following lemma.

Lemma 1. *Let Prog be a program in Ptolemy and S be an expression of Prog. Let Γ be a type environment that types S. Suppose $\Gamma \vdash \{P\}S\{Q\}$ is provable in Ptolemy. Then there is a PtolemyRely program Prog′ in which $\Gamma \vdash \{P\}S\{Q\}$ is provable by the rules for PtolemyRely.*

Proof: The new program *Prog′* in PtolemyRely is constructed by taking each event declaration E declared in *Prog*, and producing a new event declaration E' which is just like E, except that a **`relies`** clause is inserted of the form

`relies requires` P_e **`ensures`** Q_e

where $(P_e, A, Q_e) = ptolemySpec(E)$. Then the rest of the proof proceeds by induction on the structure of S.

If S is not an **`invoke`** or **`announce`** expression, then the proof rules for PtolemyRely are the same as for Ptolemy, so there are only two interesting cases.

When S is an **`invoke`** expression of the form **`next.invoke()`** then, by hypothesis, we have in Ptolemy's proof system $\Gamma \vdash \{P\}$**`next.invoke()`**$\{Q\}$. Thus by the Ptolemy (INVOKE) rule, we must have $\Gamma($**`next`**$) = closure\ Evt$, for some event name Evt, where $(P, A, Q) = ptolemySpec(Evt)$. By construction of $Prog'$, we have $(P, Q, P, A, Q) = eventSpec(Evt)$, so P plays the role of both P_e and P_r in PtolemyRely's (RINVOKE) rule, and Q plays the role of both Q_e and Q_r in that rule. So we have $\Gamma \vdash \{P \wedge P\}$**`next.invoke()`**$\{Q \vee Q\}$. To do this we use the rule of consequence in Hoare logic, since $(P \wedge P) \equiv P$ and $(Q \vee Q) \equiv Q$, to get the desired conclusion in the proof system for PtolemyRely.

When S is an **`announce`** expression of the form **`announce`** *Evt*(***y***) **`{`**S_0**`}`**, then using Ptolemy's (ANNOUNCE) rule we have: $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$, and so we also have $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S_0\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$, where Γ is the type environment for expression S, $(P_{Evt}, A, Q_{Evt}) = ptolemySpec(Evt)$ and $\boldsymbol{x} : \boldsymbol{T} = formals(Evt)$. Using PtolemyRely's (RANNOUNCEHAS) or (RANNOUNCENONE) rules, we must prove that: $\Gamma \vdash \{P_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}S\{Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}]\}$. Since by construction of *Prog′* we have that $(P_{Evt}, Q_{Evt}, P_{Evt}, A, Q_{Evt}) = eventSpec(Evt)$, then P_{Evt} plays the role of P_e and P_r, and Q_{Evt} plays the role of Q_e and Q_r, and so both rules allows us to immediately prove the desired conclusion. One can apply whichever rule is appropriate, or a derived rule with precondition $P_{Evt}[\boldsymbol{y}/\boldsymbol{x}] \wedge P_r[\boldsymbol{y}/\boldsymbol{x}]$ and postcondition $Q_{Evt}[\boldsymbol{y}/\boldsymbol{x}] \vee Q_r[\boldsymbol{y}/\boldsymbol{x}]$, and then use the rule of consequence. ∎

4 Related Work

The original work on Ptolemy [12] addressed the problem of modular reasoning of implicit invocation systems, like AO systems. Many other solutions have also been proposed: XPIs [16], AAI [10], Open Modules [1, 11], Join Point Types (JPT) [15] and Joint Point Interfaces (JPI) [8, 5, 4]. In this work we call attention to the mutual dependency that exists between the base code (*subject*) and the advising code (*handlers*). We enhanced Ptolemy's event type specifications by clearly separating the obligations imposed on the handlers from the obligations of the announced code, in such a way that both can be reasoned about modularly. Here we review how, if at all, this problem is addressed in the other approaches and if our strategy can be applied to them.

Previous work [2] has shown how the translucid contract concept of Ptolemy can be adapted to other approaches like XPIs, AAI and Open Modules; adding specification and verification capability to them. All these approaches would benefit from our enhancement to the translucid contract concept, in case they adopted it, as they would become more complete and more flexible.

Steimann *et al.* [15] proposed an approach for dealing with Implicit Invocation and Implicit Announcement (IIIA) based on Join Point Types and polymorphic pointcuts. Ptolemy's approach [3], which we extended in this work, is similar to the work of Steimann et al. One important difference, though, is that Ptolemy does not support implicit announcement. On the other hand, Steimann et al. do not treat the issue of specification and verification, suggesting that one can "resort to an informal description of the nature of the join points" [15, p. 9]. Nevertheless, since the IIIA *joinpointtype* concept is very close to the *event* concept of Ptolemy, the translucid contract approach, including our contribution, could be partially applied to join point types.

Joint Point Interfaces (JPI) [8, 5] and Closure Joint Points [4] extend and refine the notion of join point types of Steimann *et al.* JPI decouples aspects from base code and provides modular type-checking. Implicit announcement is supported through pointcuts, and explicit announcement through closure join points. JPI, similarly to JPT, lacks specification and verification features. Thus, it could also benefit from the specification and verification approach in Ptolemy and PtolemyRely.

Khatchadourian and Soundarajan [9] proposed an adaptation of the *rely-guarantee* approach used in concurrency, to be applied in aspect orientation. The base code reasoning relies on certain constraints imposed on any applicable advice. These constraints are expressed as a *rely* relation between two states. A conforming piece of advice may only make changes to the state in a way that satisfies the *rely* relation. In this way the reasoning of the base code is stable even in the presence of advice. The event pre-postconditions (P_e, Q_e) that Ptolemy imposes on every handler can be thought as a realization of the *rely* relation: $rely(\sigma_1, \sigma_2) \equiv P_e(\sigma_1) \wedge Q_e(\sigma_1, \sigma_2)$. As observed by those authors, the relation between the base code and the advice is not symmetric, as it is in the case of peer parallel processing. In their approach the base code should just *guarantee* the preconditions required by the advice. PtolemyRely follows a similar strategy, in which the base code guarantees (to the handlers) only the preconditions of the handlers. Thus in PtolemyRely: $guar(\sigma_1, \sigma_2) \equiv P_e(\sigma_1)$. Our key observation in PtolemyRely is that the advice code *might* depend on the piece of base code announced at a given join point, which may be eventually invoked from inside the advice. In PtolemyRely

we take ideas from both approaches, Ptolemy and *rely-guarantee*, and declare, as part of the event type, the conditions the advice code relies on, which corresponds to what the base code should *guarantee* to every applicable advice.

5 Conclusions and Future Work

When reasoning about event announcement in AO systems, there exists a mutual dependency between the base code (*subject*) and the advising code (*handlers*). The approach followed in systems like Ptolemy [3], where the same *requires-ensures* obligation is applied to both the handlers and the announced code, limits the flexibility and the completeness of the system.

In this paper we showed an extension to the event type concept in the Ptolemy language that explicitly separates the specification and verification of these obligations. We implemented our proposal as an extension to the Ptolemy compiler and showed that the resulting methodology is more flexible and complete than the original.

We also showed how to make the verification of the handlers more concise. Instead of verifying each handler to satisfy the event pre-post specification, one can verify, once and for all, the translucid contract of the event to satisfy this pre-post specification. Then each handler can be verified to structurally refine this translucid contract. This indirectly guarantees the required behavior of the handlers.

Previous work [2] has shown how the translucid contract concept of Ptolemy can be adapted to other approaches like XPI, AAI and Open Modules; adding specification and verification capability to them. Our work suggests that these approaches, and others like JPT and JPI, would benefit from our enhancement to the translucid contract concept.

Since event subtyping has been recently proposed for Ptolemy [6], a natural future extension to our work would be to apply the added relies clause in the presence of event polymorphism, and to analyse its impact regarding modular reasoning. We also plan to apply our approach to more complex cases, and also to use static checking techniques in the verification process.

Acknowledgments

The work of both authors was partially supported by NSF grant CCF-1017334. The work of José Sánchez is also supported by Costa Rica's Universidad Nacional (UNA), Ministerio de Ciencia y Tecnología (MICIT) and Consejo Nacional para Investigaciones Científicas y Tecnológicas (CONICIT).

References

1. Jonathan Aldrich. Open modules: Modular reasoning about advice. In Andrew P. Black, editor, *ECOOP 2005 — Object-Oriented Programming 19th European Conference, Glasgow, UK*, volume 3586 of *Lecture Notes in Computer Science*, pages 144–168. Springer-Verlag, Berlin, July 2005.

2. Mehdi Bagherzadeh, Hridesh Rajan, and Gary T. Leavens. Translucid contracts for aspect-oriented interfaces. In *FOAL '10: Workshop on Foundations of Aspect-Oriented Languages workshop*, pages 5–14, March 2010.
3. Mehdi Bagherzadeh, Hridesh Rajan, Gary T. Leavens, and Sean Mooney. Translucid contracts: expressive specification and modular verification for aspect-oriented interfaces. In *Proceedings of the tenth international conference on Aspect-oriented software development*, AOSD '11, pages 141–152. ACM, 2011.
4. Eric Bodden. Closure Joinpoints: Block joinpoints without surprises. In *AOSD '11: Proceedings of the 10th International Conference on Aspect-oriented Software Development*, pages 117–128. ACM, March 2011.
5. Eric Bodden, Éric Tanter, and Milton Inostroza. Joint point interfaces for safe and flexible decoupling of aspects. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2013. To appear.
6. Rex D. Fernando, Robert Dyer, and Hridesh Rajan. Event type polymorphism. In *Proceedings of the eleventh workshop on Foundations of Aspect-Oriented Languages*, FOAL '12, pages 33–38. ACM, 2012.
7. C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, October 1969.
8. Milton Inostroza, Éric Tanter, and Eric Bodden. Join point interfaces for modular reasoning in aspect-oriented programs. In *ESEC/FSE '11: Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 508–511, 2011.
9. Raffi Khatchadourian and Neelam Soundarajan. Rely-guarantee approach to reasoning about aspect-oriented programs. In *SPLAT '07: Proceedings of the 5th workshop on Engineering properties of languages and aspect technologies*, page 5, Vancouver, British Columbia, Canada, 2007. ACM Press.
10. Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *Proc. of the 27th International Conference on Software Engineering*, pages 49–58. ACM, 2005.
11. Neil Ongkingco, Pavel Avgustinov, Julian Tibble, Laurie Hendren, Oege de Moor, and Ganesh Sittampalam. Adding open modules to AspectJ. In *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, pages 39–50, March 2006.
12. Hridesh Rajan and Gary T. Leavens. Ptolemy: A language with quantified, typed events. In Jan Vitek, editor, *ECOOP 2008 – Object-Oriented Programming: 22nd European Conference, Paphos, Cyprus*, volume 5142 of *Lecture Notes in Computer Science*, pages 155–179, Berlin, July 2008. Springer-Verlag.
13. Steve M. Shaner, Gary T. Leavens, and David A. Naumann. Modular verification of higher-order methods with mandatory calls specified by model programs. In *International Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA), Montreal, Canada*, pages 351–367, New York, NY, October 2007. ACM.
14. J. M. Spivey. *Understanding Z: a Specification Language and its Formal Semantics*. Cambridge University Press, New York, NY, 1988.
15. Friedrich Steimann, Thomas Pawlitzki, Sven Apel, and Christian Kästner. Types and modularity for implicit invocation with implicit announcement. *ACM Trans. Softw. Eng. Methodol.*, pages 1:1–1:43, 2010.
16. Kevin Sullivan, William G. Griswold, Hridesh Rajan, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, and Nishit Tewari. Modular aspect-oriented design with xpis. *ACM Trans. Softw. Eng. Methodol.*, pages 5:1–5:42, September 2010.