Ptolemy: A Language with Quantified, Typed Events[⋆]

Hridesh Rajan and Gary T. Leavens

Iowa State University
hridesh@cs.iastate.edu

University of Central Florida
leavens@eecs.ucf.edu

Abstract. Implicit invocation (II) and aspect-oriented (AO) languages provide related but distinct mechanisms for separation of concerns. II languages have explicitly announced events that run registered observer methods. AO languages have implicitly announced events that run method-like but more powerful advice. A limitation of II languages is their inability to refer to a large set of events succinctly. They also lack the expressive power of AO advice. Limitations of AO languages include potentially fragile dependence on syntactic structure that may hurt maintainability, and limits on the available set of implicit events and the reflective contextual information available. Quantified, typed events, as implemented in our language Ptolemy, solve all these problems. This paper describes Ptolemy and explores its advantages relative to both II and AO languages.

1 Introduction

For temperance and courage are destroyed both by excess and defect, but preserved by moderation. – Aristotle, Nicomachean Ethics

The objective of both implicit invocation (II) [1–6] and aspect-oriented (AO) [7] languages is to improve a software engineer's ability to separate conceptual *concerns*. The problem that they address is that not all concerns are amenable to modularization by a single dimension of decomposition [8]; instead, some concerns cut across the main dimension of decomposition. For example, code implementing a visualization concern would be scattered across the classes of an object-oriented (OO) decomposition. The II and AO approaches aim to better encapsulate such crosscutting concerns and decouple them from other code, thereby easing maintenance.

However, both II and AO languages suffer from various limitations. The goal of this paper is to explain how our language Ptolemy, which combines the best ideas of both kinds of language, can solve many of these problems.

1.1 Implicit Invocation Languages and their Limitations

The key idea in II languages is that *events* are used as a way to interface two sets of modules, so that one set can be independent of the other. Events promote decoupling and can be seen as direct linguistic support for the Observer pattern [9]. The mechanisms of an II language are also known as "event subscription management." [3]

⋆ Rajan was supported in part by the NSF grant CNS-0627354. Leavens was supported in part by NSF grant CCF-0429567. Both were supported in part by NSF grant CNS-07-09217.

```
1 abstract class FElement extends Object{
2  event ChangeEvent(FElement changedFE);
3  event MoveUpEvent(FElement targetFE,
4       Number y, Number delta);
5 }
6 class Point extends FElement { /* ... */
7  Number x; Number y;
8  FElement setX(Number x) {
9   this.x = x;
10  announce ChangeEvent(this);
11  this
12 }
13 FElement moveUp(Number delta) {
14  announce MoveUpEvent(this,this.y,delta);
15  this.y = this.y.plus(delta); this
16 }
17 FElement makeEqual(Point other) {
18  other.x = this.x; other.y = this.y;
19  announce ChangeEvent(other); other
20 }
21 }
```

```
22 class Update extends Object { /* ... */
23  FElement last;
24  Update registerWith(FElement fe) {
25   fe.register(this, FElement.ChangeEvent);
26   fe.register(this, FElement.MoveUpEvent);
27   this
28  }
29  FElement update(FElement changedFE, Number x){
30   this.last = changedFE;
31   Display.update();
32   changedFE
33  }
34  FElement check(FElement targetFE,
35     Number y, Number delta) {
36   if (delta.lt(100)) { changedFE }
37   else{throw new IllegalArgumentException()}
38  }
39  when FElement.ChangeEvent do update
40  when FElement.MoveUpEvent do check
41 }
```

Fig. 1. Drawing Editor in an II language.

With declared events, certain modules (subjects) dynamically and explicitly *announce* events. Another set of modules (observers) can dynamically *register* methods, called *handlers*. These handlers are invoked (implicitly) when events are announced. The subjects are thus independent of the particular observers.

Figure 1 illustrates the mechanisms of a hypothetical Java-like II language based on Classic Java (and thus similar to Ptolemy) for a figure editor that we will use as a running example in this paper. This code is part of a larger editor that works on drawings comprising points, lines, and other such figure elements [10, 11]. The code announces two kinds of events, named `ChangeEvent` and `MoveUpEvent` (lines 2–4). The subclass `Point` announces these events using **`announce`** expressions (lines 10, 14, and 19). When an instance of the class `Update` is properly initialized, by calling the `registerWith` method on an instance of the `Point` class, these announcements will implicitly invoke the methods of class `Update` (lines 22–41). The connection between the events and methods of class `Update` is made on lines 39–40, where it is specified that the `update` method is to be called when the `ChangeEvent` occurs and the `check` method when `MoveUpEvent` occurs. Dynamic registration (lines 25–26) allows the receiver of these method calls to be determined (and allows unregistration and multiple registration).

The main advantage of an II language over OO languages is that it provides considerable automation of the Observer pattern [3], which is key to decoupling subject modules from observer modules. That is, modules that announce events remain independent of the modules that register methods to handle their event announcements. Compared to AO languages, as we will see, II languages also have some advantages. First, event announcement is explicit, which helps in understanding the module announcing the event, since the points where events may occur are obvious from the code. Second, event announcement is flexible; i.e., arbitrary points in the program can be exposed as events.

However, compared with AO languages, II languages also have three limitations: coupling of observers to subjects, no ability to replace event code, and lack of quantification. We describe these below.

Coupling of Observers to Subjects While subjects need not know about observers in an II language, the observer modules still know about the subjects. In Figure 1, for example, the registration code on lines 25–26 and the binding code on lines 39–40 mentions the events declared in `FElement`. (Mediators, a design style for II languages, also decouple subjects and observers so that they can be changed independently [6]. However, mediator modules remain coupled to both the subject and observers.)

No Replacement of Event Code The ability to replace the code for an event (what AO calls “around advice”), is not available, without unnecessarily complex emulation code (to simulate closures in languages such as Java and C#). Instead, to stop an action, one must have a handler throw an exception (as in line 37), which does not clearly express the idea. Similarly, throwing an exception does not support replacing actions with different actions, such as replacing a local method call with a remote method invocation.

No Quantification In II languages describing how each event is handled, which following the AO terminology we call *quantification*, can be tedious. Indeed, such code can grow in proportion to the number of objects from which implicit invocations are to be received. For example, to register an `Update` instance `u` to receive implicit invocations when events are announced by both a point `p` and a line `l`, one would write the following code: `u.registerWith(p); u.registerWith(l)`. One can see that such registration code has to find all figure element instances. In this case these problems are not too bad, since all such instances have types that are subtypes of `FElement`, where the relevant events are declared. However, if the events were announced in unrelated classes, then the registration code (lines 25–26) and the code that maps events to method calls (lines 39–40) would be longer and more tedious to write.

1.2 Aspect-Oriented Languages and their Limitations

In AO languages [12, 13] such as AspectJ [7, 10, 14, 15] (which we emphasize for the maturity of its design and the availability of a workable implementation) events (called “join points”) are pre-defined by the language as certain kinds of standard actions (such as method calls) in a program’s execution. AO events are all implicitly announced. *Pointcut descriptions (PCDs)* are used to declaratively register handlers (called “advice”) with sets of events. Using PCDs to register a handler with an entire set of events, called *quantification* [16], is a key idea in AO languages that has no counterpart in II languages. A language’s set of PCDs and events form its *event model* (in AO terms this is a “join point model”).

The listings in Figure 2 shows an AspectJ-like implementation for the drawing editor discussed before. (We have adapted the syntax of AspectJ to be more like our language Ptolemy, to make comparisons easier.) In this implementation the `Point` class (and other figure elements such as `Line`), would be free of any event-related code. Modularization of display update is done with an aspect. This aspect uses PCDs such

```
1  abstract class FElement extends Object {}
2  class Point implements FElement { /*...*/
3   Number x; Number y;
4   FElement setX(Number x) {
5    this.x = x; this
6   }
7   FElement moveUp(Number delta) {
8    this.y = this.y.plus(delta); this
9   }
10  FElement makeEqual(Point other) {
11   other.x = this.x;
12   other.y = this.y; other
13  }
14 }
15 aspect Update {
16  FElement around(FElement fe) :
17   call(FElement+.set*(..)) && target(fe)
18   || call(FElement+.makeEq*(..)) && args(fe){
19   FElement res = proceed(fe);
20   Display.update(); res
21  }
22  FElement around(FElement fe, Number delta):
23    target(fe)&&(call(FElement+.move*(..))
24    && args(delta){
25   if (delta.lt(100) { proceed(delta) }
26   else { fe }
27  }
28 }
```

Fig. 2. Drawing editor's AO implementation.

as **target**(fe) && **call**(FElement+.set*(..)) to select events that change the state of figure elements. This PCD selects events that call a method matching `set*` on a subtype of `FElement` and binds the context variable `fe` (of type `FElement`) to that call's receiver.

AO languages also have several advantages. Quantification provides ease of use. For example, one can select events throughout a program (and bind them to handlers) by just writing a simple regular expression based PCD, as on lines 17–18. Moreover, by not referring to the names in the modules announcing events directly, the handler code remains, at least syntactically, independent of that code. Implicit event announcement both automates and further decouples the two sets of modules, compared with II languages. This property, sometimes called *obliviousness* [16], avoids the "scattering" and "tangling" [7] of event announcement code within the other code for the subjects, which can be seen in lines 10, 14, and 19 of Figure 1. In that figure, this explicit announcement code is mixed in with other code, resulting in tangled code that makes it harder to follow the main program flow.

However, AO languages suffer from four limitations, primarily because most current event models use PCDs based on pattern matching. These languages differ by what they match. For example, AspectJ-like languages use pattern matching on lexical names [10], LogicAJ and derivative languages use pattern matching on lexical structures [17, 18], and, history-based pointcuts use pattern matching on program traces [19]. An example PCD in languages that match names is **call**(FElement+.set*(..)) that describes a set of call events in which the name of the called method starts with "`set`". An example PCD in languages that match lexical structures is stmt(?**if**,**if**(?**call**){??someStatements}&&fooBarCalls(?**call**) that describes a set of call events in which the name of the called method is "`foo`" or "`bar`" and the call occurs within an **if** condition [17, Fig 4.]. An example PCD in languages that match program traces would be $G(call(*Line.set(..)) \rightarrow F(call(*Point.set(..))))$ that describes every call event in which the name of the called method is "`Line.set`" and that is finally followed by another call event in which the name of the called method is "`Point.set`" [20, Fig 3.].

Fragile Pointcuts The fragility of pointcuts results from the use of pattern matching as a quantification mechanism [21, 22]. Such PCDs are coupled to the code that implements the implicit events they describe. Thus, seemingly innocuous changes break aspects. For example, for languages that match based on names, a change such as adding new methods that match the PCD, such as `settled`, can break an aspect that is counting on methods that start with "`set`" to mean that the object is changing. As pointed out by Kellens *et al.* [23], in languages that match based on lexical structures a simple change such as changing an **`if`** statement to an equivalent statement that used a conditional (`?:`) expression would break the aspect. For languages that match based on program traces a simple change such as to inline the functionality of "`Point.Set`" would break the aspect that is counting on "`Line.Set`" to be eventually followed by "`Point.Set`" [23]. Conversely, when adding a method such as `makeEqual` that does not conform to the naming convention, one has to change the PCD to add the new method pattern (as shown in line 18 of Figure 2). In the same vein, when adding a new point such as `foo` within **`while`** that does not conform to the lexical structure, one has to change the PCD to add the new lexical structure pattern. Similar arguments apply for trace-based pointcuts. Indeed, to fix such problems PCDs must often be changed (e.g., to exclude or include added methods). Such maintenance problems can be important in real examples.

Several recent ideas such as Aspect Aware Interfaces (AAIs) [11], Crosscut Programming Interfaces (XPIs) [24, 25], Model-based Pointcuts [23], Open Modules (OM) [26], etc, have recognized and proposed to address the fragile pointcut problem. Briefly, AAIs, computed using the global system configuration, allow a developer to see the events in the base code quantified by a PCD, but do not help with reducing the impact of base code changes on PCDs, which primarily causes the fragile pointcut problem. XPIs reduce the scope of fragile pointcut problem to the scope declared as part of the interface, however, within a scope the problem remains. OMs allow a class to explicitly expose the set of events, however, for quantifying such events explicit enumeration is needed, which couples the PCD with names in the base code. Such enumerations are also potentially fragile as pointed out by Kellens *et al.* [23]. A detailed discussion of these ideas is presented in Section 4.

Quantification Failure The problem of quantification failure is caused by incompleteness in the language's event model. It occurs when the event model does not implicitly announce some kinds of events and hence does not provide PCDs that select such events [24, pp. 170]. In AspectJ-like AO languages there is a fixed classification of potential event kinds and a corresponding fixed set of PCDs. For example, some language features, such as loops or certain expressions, are not announced as events in AspectJ and have no corresponding PCDs.[1] While there are reasons (e.g., increased coupling) for not making some kinds of potential events available, some practical use cases need to handle them [27, 28]. This fixed set of event kinds and PCDs contributes to quantification failure, because some events cannot be announced or used in PCDs.

[1] Some may view that as a problem of the underlying language rather than the approach to aspects: e.g., in a language where all computation takes place in methods, this, target and args are always defined. We argue that it may not be necessary to continue to support such differentiation between means of computation, instead a unified view of all such means of computation can be provided to the aspects.

There are approaches such as LogicAJ that provide a finer-grained event model [17]. For example, in LogicAJ one could match arbitrary lexical structure in the base code, which is significantly more expressive compared to matching based on names. However, as discussed previously a problem with such technique is that the PCDs becomes strongly coupled with the structure of the base code and therefore become more fragile.

An alternative approach to solving this problem is taken by techniques such as SetPoint [29]. These techniques allow a programmer to select events by attaching annotations to locations of such events. This technique is not fragile in the sense that it does not depend on lexical names, structure, or order of events. A problem, however, is that this technique does not allow arbitrary expressions to be selected, primarily because the underlying languages do not allow annotations on arbitrary expressions.

Limited Access to Context Information Current AO languages provide a limited interface for accessing contextual (or reflective) information about an event [24]. For example, in AspectJ, a handler (advice) can access only fixed kinds of contextual information from the event, such as the current receiver object (**`this`**), a call's target, its arguments, etc. Again there are good reasons for limiting this interface (e.g., avoiding coupling), but the fundamental problem is that, in current languages, this interface is fixed by the language designer and does not satisfy all usage scenarios. For example, when modularizing logging, developers need access to the context of the logging events, including local variables. However, local variables are not available in existing AO event models.

Approaches such as LogicAJ [17] allow virtually unlimited reflective access to the context surrounding the lexical structure using meta-variables, which is more expressive compared to AspectJ's model; e.g., a local variable can be accessed by associating it with a meta-variable. However, as we discuss in detail below, this unlimited access is achieved with ease only in cases where the events form a regular structure.

Uniform Access to Irregular Context Information A related problem is when contextual information that fulfills a common need (or role) in the handlers is not available uniformly to PCDs (and handlers). For example, in Figure 2 `setX` and `makeEqual` contribute to the event "changing a figure element," however, they are changing different figure element instances: **`this`** and `other` in the case of `setX` and `makeEqual` respectively. In this simple case, it is possible to work around this issue by writing a PCD that combines (using `||`, as in lines 17–18 of Figure 2) two separate PCDs, as shown in Figure 2. Each of these PCDs accesses the changed instance differently (one using **`target`**, the other using **`args`**). However, each such PCD depends on the particular code features that it needs to access the required information.

This problem is present in even significantly more expressive approaches based on pattern matching such as LogicAJ [17]. For irregular context information, the best solution in these techniques also need to resort to explicit enumeration of base code structure to identify meta-information that need to be accessed. Note that such enumeration increasing the coupling between the PCDs and the details of the base code.

1.3 Contributions

In this work, we present a new language, Ptolemy, which adds quantified, typed events to II languages, producing a language that has many of the advantages of both II and AO languages, but suffers from none of the limitations described above.

Ptolemy declares named event types independently from the modules that announce or handle these events. These event types provide an interface that completely decouples subject and observer modules. An event type p also declares the types of information communicated between announcements of events of type p and handler methods. Events are explicitly announced using **`event`** expressions. Event expressions enclose a body expression, which can be replaced by a handler, providing expressiveness akin to **`around`** advice in AO languages. Event type names can also be used in quantification, which simplifies binding and avoids coupling observers with subjects.

Key differences between Ptolemy and II languages are thus:

- separating event type declarations from the modules that announce events,
- the ability to treat an expression's execution as an event,
- the ability to override that execution, and
- quantification by the use of PCDs.

Key differences between Ptolemy and AO languages are:

- events are explicitly announced, but quantification over them does not require enumeration unlike techniques such as Open Modules [26],
- an arbitrary expression can be identified as an event (unlike Setpoint [29]) without exacerbating the fragile pointcut problem (unlike languages like LogicAJ [17]),
- events can communicate an arbitrary set of reflective information to handlers without coupling handlers to the lexical details (cf. [23]), and
- PCDs can use declared event types for quantification.

The benefit of Ptolemy's new features over II languages is that the separation of event type declarations allows further decoupling, and that the ability to replace events completely is more powerful. The benefit over AO languages is that handler methods (advice) can uniformly access reflective information from the context of events without breaking encapsulation of the code that announces events. Furthermore, event types also permit further decoupling over AO languages, since PCDs are decoupled from the code announcing events (the "base code").

These benefits make Ptolemy an interesting point in the design space between II and AO languages. Since event announcement is explicit, announcing modules are not completely "oblivious" to the presence of handlers, and hence by some definitions [16] Ptolemy is not aspect-oriented. However, this lack of obliviousness is not fatal for investigating its utility as a language design, and indeed highlights the advantages and disadvantages of obliviousness, as we will explain below.

In summary, this work makes the following contributions. It presents:

- a language design with simple and flexible event model;
- a precise operational semantics and type system for the language's novel constructs;
- an implementation of the language as an extension of Eclipse's Java compiler; and,
- a detailed analysis of our approach and the closely related ideas.

2 Ptolemy's Design

Ptolemy (Claudius Ptolemaeus), fl. 2d cent. A.D.,
celebrated Greco-Egyptian mathematician, astronomer, and geographer.

In this section, we describe Ptolemy's design. Its use of quantified, typed events extends II languages with ideas from AO languages. Ptolemy features new mechanisms for declaring event types and events. It is inspired by II languages such as Rapide [3] and AO languages such as AspectJ [10]. It also incorporates some ideas from Eos [30] and Caesar [31]. As a small, core language, its technical presentation shares much in common with MiniMAO$_1$ [32, 33]. The object-oriented part of Ptolemy has classes, objects, inheritance, and subtyping, but it does not have **`super`**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods. The novel features of Ptolemy are found in its event model and type system.

Like Eos [30], Ptolemy does not have special syntax for "aspects" or "advice". Instead it has the capability to replace all events in a specified set (a pointcut) with a call to a handler method. Each handler takes a proceed closure as its first argument. A *proceed closure* [30] contains code needed to run the applicable handlers and the original event's code. A proceed closure is run by a **`proceed`** expression.

Like II languages a class in Ptolemy can register handlers for events. However, unlike II languages, where one has to write an expression for registering a handler with each event in a set, Ptolemy allows a handler to be declaratively registered for a set of events using one succinct PCD in a *binding* (which is similar to declaring AO "around advice"). At runtime, one can use Ptolemy's **`register`** expression to activate such relationships. The **`register`** expression supplies an *observer instance* (an object) that becomes the receiver in calls to its handler methods that are made when the corresponding events are announced.[2] It is thus easy to make individual observer instances that handle event announcements ("instance-level advising") [34]. Singleton "aspects" could be easily added as syntactic sugars.

2.1 Syntax

Ptolemy's syntax is shown in Figure 3 and explained below. A program in Ptolemy consists of a sequence of declarations followed by an expression. The expression can be thought of as the body of a "main" method. In Figure 4 we illustrated the syntax using the example from Section 1.

Declarations The two top-level declaration forms, classes and event type declarations, may not be nested. A class has exactly one superclass, and may declare several fields, methods, and bindings. Bindings associate a handler method to a set of events described by a pointcut description (PCD). The binding in Figure 4, line 47 says to run `update` when events of type `FEChange` are announced. Similarly, the binding on line 48 says to run `check` when events of type `MoveUpEvent` are announced.

[2] In AO languages such as Eos [34] and Caesar [31] register expressions correspond to "deploying aspects."

```
prog ::= decl* e
decl ::= c evtype p { form* }
      | class c extends d { field* meth* binding* }
field ::= c f;
meth ::= t m (form*) { e }
t ::= c | thunk c
binding ::= when pcd do m
form ::= t var,      where var≠this
pcd ::= p | pcd '||' pcd
e ::= new c() | var | null | e.m(e*) | e.f
     | e.f = e | cast c e | form = e; e | e; e
     | register(e) | event p { e } | proceed(e)
```

where
$c, d \in \mathcal{C}$, a set of class names
$p \in \mathcal{P}$, a set of evtype names
$f \in \mathcal{F}$, a set of field names
$m \in \mathcal{M}$, a set of method names
$var \in \{\texttt{this}\} \cup \mathcal{V}, \mathcal{V}$ is a set of variable names

Fig. 3. Ptolemy's abstract syntax, based on Clifton's dissertation [32, Figures 3.1, 3.7].

```
1 FElement evtype FEChange{
2  FElement changedFE;
3 }
4 FElement evtype MoveUpEvent{
5  FElement targetFE; Number y; Number delta;
6 }
7 class FElement extends Object{}
8 class Point extends FElement{ /* ... */
9  Number x; Number y;
10  FElement setX(Number x) {
11   FElement changedFE = this;
12   event FEChange{ this.x = x; this }
13  }
14  FElement moveUp(Number delta){
15   FElement movedFE = this;
16   event MoveUpEvent{
17    this.y = this.y.plus(delta); this
18   }
19  }
20  FElement makeEqual(Point other){
21   FElement changedFE = other;
22   event FEChange{
23    other.x = this.x;
24    other.y = this.y; other
25    }
26  }
27 }
```

```
28 class Update extends Object{
29  FElement last;
30  Update init(){
31   register(this)
32  }
33  FElement update(thunk FElement next,
34      FElement changedFE){
35   FElement res = proceed(next);
36   this.last = changedFE;
37   Display.update(); res
38  }
39  FElement check(thunk FElement next,
40        FElement targetFE,
41        Number y, Number delta){
42   if (delta.lt(100)){
43    FElement res = proceed(next)
44   };
45   targetFE
46  }
47  when FEChange do update
48  when MoveUpEvent do check
49 }
```

Fig. 4. Drawing Editor in Ptolemy

An event type (**`evtype`**) declaration has a return type (*c*), a name (*p*), and zero or more context variable declarations (*form**). These context declarations specify the types and names of reflective information exposed by conforming events. Two examples are given in Figure 4 on lines 1–6. The intention of the first event type declaration (lines 1–3) is to provide a named abstraction for a set of events, with result type `FElement`, that contribute to an abstract state change in a figure element, such as moving a point. This event type declares only one context variable, `changedFE`, which denotes the `FElement` instance that is being changed. Similarly, the event type `MoveUpEvent` (lines 4–6) declares three context variables, `targetFE`, which denotes the `FElement` instance that is moving up, `y`, the current Cartesian co-ordinate value for that instance, and `delta`, the displacement of the instance.

Quantification: Pointcut Descriptions PCDs have two forms. The named PCD denotes the set of events identified by the programmer using event expressions with the given name. Two examples appear on lines 47–48 of Figure 4. The first, `FEChange`, denotes events identified with the type `FEChange`. The context exposed by this PCD is the subset of the lexical context named by that event type and available at event expressions that mention that type.

The disjunction (`||`) of two PCDs gives the union of the sets of events denoted by the two PCDs. The context exposed by the disjunction is the intersection of the context exposed by the two PCDs. However, if an identifier I is bound in both contexts, then I's value in the exposed context is I's value from the right hand PCD's context.

Expressions Ptolemy is an expression language, thus the syntax for expressions includes several standard object-oriented (OO) expressions [32, 33, 35].

There are three new expressions: **`register`**, **`proceed`**, and **`event`**. The expression **`register`**`(`e`)` evaluates e to an object o, registers o by putting it into the program's list of active objects, and returns o. The list of active objects is used in the semantics to track registered objects. Only objects in this list are capable of advising events. For example lines 30–32 of Figure 4 is a method that, when called, will register the method's receiver (**`this`**). The expression **`proceed`**`(`e`)` evaluates e, which must denote a proceed closure, and runs that proceed closure. This runs the handler in the proceed closure or, if there are no handlers, the proceed closure's original expression.

The expression **`event`** p `{`e`}` announces e as an event of type p and runs any handlers of registered objects that are applicable to p, using a registered object as the receiver and passing as the first argument a proceed closure. This proceed closure contains the rest of the handlers and the original expression e and its lexical environment. In Figure 4 the event expression on line 10 has a body consisting of a sequence expression. Notice that the body of the `setX` method contains a block expression, where the definition on line 11 binds **`this`** to `changedFE`, and then evaluates its body, the event expression. This definition makes the value of **`this`** available in the variable `changedFE`, which is needed by the context declared for the event type `FEChange`. In this figure, the event declared on line 22–25 also encloses a sequence expression. As required by the event type, the definition on line 21 of Figure 4 makes the value of `other` available in the variable `changedFE`. Thus the first and the second event expressions are given different bindings for the variable `changedFE`, however, code that advises this event type will be able to access this context uniformly using the name `changedFE`.

The II syntax "**`announce`** p" can be thought of as sugar for "**`event`** p `{`**`null`**`}`." Thus Ptolemy's event announcement is strictly more powerful than that in II languages.

2.2 Operational Semantics of Ptolemy

This section defines a small step operational semantics for Ptolemy. The semantics is based on Clifton's work [32, 33, 36], which builds on Classic Java [37].

The expression semantics relies on four expressions that are not part of Ptolemy's surface syntax as shown in Figure 5. The *loc* expression represents locations in the store. The **`under`** expression is used as a way to mark when the evaluation stack needs popping. The two exceptions record various problems orthogonal to the type system.

Added Syntax:

$e ::= loc \mid$ **`under`** $e \mid$ `NullPointerException` $\mid$ `ClassCastException`
where $loc \in \mathcal{L}$, a set of locations

Domains:

$\Gamma ::= \langle e, J, S, A\rangle$	"Configurations"
$J ::= \nu + J \mid \bullet$	"Stacks"
$\nu ::=$ **`lexframe`** $\rho\ \Pi \mid$ **`evframe`** $p\ \rho\ \Pi$	"Frames"
$\rho ::= \{j : v_k\}_{k \in K}$, **where** K is finite, $K \subseteq I$	"Environments"
$v ::= loc \mid$ **`null`**	"Values"
$S ::= \{loc_k \mapsto sv_k\}_{k \in K}$, **where** K is finite	"Stores"
$sv ::= o \mid pc$	"Storable Values"
$o ::= [c.F]$	"Object Records"
$F ::= \{f_k \mapsto v_k\}_{k \in K}$, **where** K is finite	"Field Maps"
$pc ::=$ **`pClosure`**$(H, \theta)\,(e, \rho, \Pi)$	"Proceed Closures"
$H ::= h + H \mid \bullet$	"Handler Record Lists"
$h ::= \langle loc, m, \rho'\rangle$	"Handler Records"
$A ::= loc + A \mid \bullet$	"Active (Registered) List"

Evaluation contexts:

$\mathbb{E} ::= - \mid \mathbb{E}.m(e\ldots) \mid v.m(v\ldots\mathbb{E}\,e\ldots) \mid$ **`cast`** $t\ \mathbb{E} \mid \mathbb{E}.f \mid \mathbb{E};e \mid \mathbb{E}.f\texttt{=}e$
$\mid v.f\texttt{=}\mathbb{E} \mid t\ var\texttt{=}\mathbb{E};\ e \mid \mathbb{E};\ e \mid$ **`register`**$(\mathbb{E}) \mid$ **`under`** $\mathbb{E} \mid$ **`proceed`**$(\mathbb{E})$

Fig. 5. Added syntax, domains, and evaluation contexts used in the semantics, based on [32].

Figure 5 also describes the configurations, and the evaluation contexts in the operational semantics, most of which is standard and self-explanatory. A configuration contains an expression (e), a stack (J), a store (S), and an ordered list of active objects (A). Stacks are an ordered list of frames, each frame recording the static environment (ρ) and some other information. (The type environments Π are only used in the type soundness proof [35].) There are two types of stack frame. Lexical frames (**`lexframe`**) record an environment ρ that maps identifiers to values. Event frames (**`evframe`**) are similar, but also record the name p of the event type being run. Storable values are objects or proceed closures. Proceed closures (**`pClosure`**) contain an ordered list of handler records (H), a PCD type (θ), an expression (e), an environment (ρ), and a type environment (Π). The type θ and the type environment Π (see Figure 7) are maintained by but not used by the operational semantics; they are only used in the type soundness proof [35]. Each handler record (h) contains the information necessary to call a handler method: the receiver object (loc), a method name (m), and an environment (ρ'). The environment ρ' is used to assemble the method call arguments when the handler method is called. The environment ρ recorded at the top level of the proceed closure is used to run the expression e when a proceed closure with an empty list of handler records is used in a **`proceed`** expression.

Figure 6 presents the key rules. The details about standard OO rules are omitted here, however, interested reader can refer to our technical report on Ptolemy [35]. The rules all make implicit use of a fixed (global) list, CT, of the program's declarations.

The (EVENT) rule is central to Ptolemy's semantics, as it starts the running of handler methods. In essence, the rule forms a new frame for running the event, and then looks up bindings applicable to the new stack, store, and list of registered (active) objects. The resulting list of handler records (H) is put into a proceed closure

Evaluation relation: $\hookrightarrow : \mathit{\Gamma} \to \mathit{\Gamma}$

(EVENT)

$$\frac{\begin{array}{c}\rho = \mathit{envOf}(\nu) \qquad \mathit{\Pi} = \mathit{tenvOf}(\nu) \qquad (c\ \mathbf{evtype}\ p\{t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n\}) \in CT \\ \rho' = \{\mathit{var}_i \mapsto v_i \mid \rho(\mathit{var}_i) = v_i\} \qquad \pi = \{\mathit{var}_i : \mathbf{var}\ t_i \mid 1 \le i \le n\} \\ \mathit{loc} \notin \mathit{dom}(S) \qquad \pi' = \pi \uplus \{\mathit{loc} : \mathbf{var}\ (\mathbf{thunk}\ c)\} \qquad \nu' = \mathbf{evframe}\ p\ \rho'\ \pi' \\ H = \mathit{hbind}(\nu' + \nu + J, S, A) \qquad \theta = \mathbf{pcd}\ c, \pi \qquad S' = S \oplus (\mathit{loc} \mapsto \mathbf{pclosure}(H, \theta)\ (e, \rho, \mathit{\Pi}))\end{array}}{\langle \mathbb{E}[\mathbf{event}\ p\ \{e\}], \nu + J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ (\mathbf{proceed}(\mathit{loc}))], \nu' + \nu + J, S', A\rangle}$$

(UNDER)

$$\begin{array}{l}\langle \mathbb{E}[\mathbf{under}\ v], \nu + J, S, A\rangle \\ \quad \hookrightarrow \langle \mathbb{E}[v], J, S, A\rangle\end{array}$$

(REGISTER)

$$\begin{array}{l}\langle \mathbb{E}[\mathbf{register}(\mathit{loc})], J, S, A\rangle \\ \quad \hookrightarrow \langle \mathbb{E}[\mathit{loc}], J, S, \mathit{loc} + A\rangle\end{array}$$

(PROCEED-DONE)

$$\frac{\mathbf{pclosure}(\bullet, \theta)\ (e, \rho, \mathit{\Pi}) = S(\mathit{loc}) \qquad \nu = \mathbf{lexframe}\ \rho\ \mathit{\Pi}}{\langle \mathbb{E}[\mathbf{proceed}(\mathit{loc})], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ e], \nu + J, S, A\rangle}$$

(PROCEED-RUN)

$$\frac{\begin{array}{c}\mathbf{pclosure}((\langle \mathit{loc}', m, \rho\rangle + H), \theta)\ (e, \rho', \mathit{\Pi}) = S(\mathit{loc}) \\ [c.F] = S(\mathit{loc}') \qquad (c_2, t\ m(t_1\mathit{var}_1, \ldots, t_n\mathit{var}_n)\{e'\}) = \mathit{methodBody}(c, m) \\ n \ge 1 \qquad \rho'' = \{\mathit{var}_i \mapsto v_i \mid 2 \le i \le n, v_i = \rho(\mathit{var}_i)\} \qquad \mathit{loc}_1 \notin \mathit{dom}(S) \\ S' = S \oplus (\mathit{loc}_1 \mapsto \mathbf{pclosure}(H, \theta)\ (e, \rho', \mathit{\Pi})) \qquad \rho''' = \rho'' \oplus \{\mathit{var}_1 \mapsto \mathit{loc}_1\} \oplus \{\mathbf{this} \mapsto \mathit{loc}'\} \\ \mathit{\Pi}' = \{\mathit{var}_i : \mathbf{var}\ t_i \mid 1 \le i \le n\} \uplus \{\mathbf{this} : \mathbf{var}\ c_2\} \qquad \nu = \mathbf{lexframe}\ \rho'''\ \mathit{\Pi}'\end{array}}{\langle \mathbb{E}[\mathbf{proceed}(\mathit{loc})], J, S, A\rangle \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ e'], \nu + J, S', A\rangle}$$

Fig. 6. Operational semantics of Ptolemy, based on [32]. Standard OO rules are omitted.

($\mathbf{pclosure}(H, \theta)\ (e, \rho', \mathit{\Pi})$)), which is placed in the store at a fresh location. This proceed closure will execute the handler methods, if any, before the body of the event expression (e) is evaluated. Since a new (event) frame is pushed on the stack, the **proceed** expression that starts running this closure is placed in an **under** expression. The (UNDER) rule pops the stack when evaluation of its subexpression is finished.

The auxiliary function hbind [35] uses the program's declarations, the stack, store, and the list of active objects to produce a list of handler records that are applicable for the event in the current state. When called by the (EVENT) rule, the stack passed to it has a new frame on top that represents the current event.

The (REGISTER) rule simply puts the object being activated at the front of the list of active objects. The bindings in this object are thus given control before others already in the list. An object can appear in this list multiple times.

The evaluation of **proceed** expressions is done by the two proceed rules. The (PROCEED-DONE) rule handles the case where there are no (more) handler records. It simply runs the event's body expression (e) in the environment (ρ) that was being remembered for it by the proceed closure.

The (PROCEED-RUN) rule handles the case where there are handler records still to be run in the proceed closure. It makes a call to the active object (referred to by loc) in the first handler record, using the method name and environment stored in that handler record. The active object is the receiver of the method call. The first formal parameter is bound to a newly allocated proceed closure that would run the rest of the handler records (and the original event's body) if it used in a **proceed** expression.

$\theta ::= \text{OK} \mid \text{OK in } c \mid \mathtt{var}\ t \mid \mathtt{exp}\ t \mid \mathtt{pcd}\ \tau, \pi$	"type attributes"
$\tau ::= c \mid \bot$	"class type exps"
$\pi, \Pi ::= \{I : \theta_I\}_{I \in K}$,	"type environments"
where K is finite, $K \subseteq (\mathcal{L} \cup \{\mathtt{this}\} \cup \mathcal{V})$	

Fig. 7. Type attributes.

2.3 Ptolemy's Type System

Type checking uses the type attributes defined in Figure 7. The type checking rules themselves are shown in Figure 8. Standard rules for OO features are omitted [35]. The notation $\tau' \preccurlyeq \tau$ means τ' is a subtype of τ. It is the reflexive-transitive closure of the declared subclass relationships [35].

As in Clifton's work [32, 33], the type checking rules are stated using a fixed class table (list of declarations) CT, which can be thought of as an implicit (hidden) inherited attribute. This class table is used implicitly by many of the auxiliary functions. For ease of presentation, we also follow Clifton in assuming that the names declared at the top level of a program are distinct and that the extends relation on classes is acyclic.

(CHECK BINDING)
$$\frac{\begin{array}{c}(c_2, c'\ m\,(t_1\ var_1, \ldots, t_n\ var_n)\,\{e\}) = methodBody(c, m) \quad \vdash pcd : \mathtt{pcd}\ c', \pi \quad isClass(c') \\ n \geq 1 \quad t_1 = \mathtt{thunk}\ c' \quad (\forall i \in \{2..n\} :: isType(t_i)) \quad \{var_2 : \mathtt{var}\ t_2, \ldots, var_n : \mathtt{var}\ t_n\} \subseteq \pi\end{array}}{\Pi \vdash (\mathtt{when}\ pcd\ \mathtt{do}\ m) : \text{OK in } c}$$

(CHECK EVTYPE)
$$\frac{isClass(c) \quad (\forall i \in \{1..n\} :: isType(t_i))}{\vdash c\ \mathtt{evtype}\ p\ \{t_1\ var_1;\ \ldots t_n\ var_n;\} : \text{OK}}$$

(EV ID PCD TYPE)
$$\frac{(c\ \mathtt{evtype}\ p\ \{t_1\ var_1;\ \ldots t_n\ var_n;\}) \in CT \quad \pi = \{var_1 : \mathtt{var}\ t_1, \ldots var_n : \mathtt{var}\ t_n\}}{\vdash p : \mathtt{pcd}\ c, \pi}$$

(DISJUNCTION PCD TYPE)
$$\frac{\vdash pcd : \mathtt{pcd}\ \tau, \pi \quad \vdash pcd' : \mathtt{pcd}\ \tau', \pi' \quad \tau'' = \tau \sqcap \tau' \quad \pi'' = \pi \cap \pi'}{\vdash pcd\ ||\ pcd' : \mathtt{pcd}\ \tau'', \pi''}$$

(UNDER EXP TYPE)
$$\frac{\Pi \vdash e : \mathtt{exp}\ t}{\Pi \vdash \mathtt{under}\ e : \mathtt{exp}\ t}$$

(REGISTER EXP TYPE)
$$\frac{\Pi \vdash e : \mathtt{exp}\ c}{\Pi \vdash \mathtt{register}(e) : \mathtt{exp}\ c}$$

(PROCEED EXP TYPE)
$$\frac{\Pi \vdash e : \mathtt{exp}\ (\mathtt{thunk}\ c)}{\Pi \vdash \mathtt{proceed}(e) : \mathtt{exp}\ c}$$

(EVENT EXP TYPE)
$$\frac{\begin{array}{c}(c\ \mathtt{evtype}\ p\ \{t_1\ var_1;\ \ldots t_n\ var_n;\}) \in CT \\ \{var_1 : \mathtt{var}\ t_1, \ldots, var_n : \mathtt{var}\ t_n\} \subseteq \Pi \quad \Pi \vdash e : \mathtt{exp}\ c' \quad c' \preccurlyeq c\end{array}}{\Pi \vdash \mathtt{event}\ p\ \{e\} : \mathtt{exp}\ c}$$

Auxiliary Functions:

$$isClass(t) = (\mathtt{class}\ t \ldots) \in CT$$
$$isThunkType(t) = (t = \mathtt{thunk}\ c \wedge isClass(c))$$
$$isType(t) = isClass(t) \vee isThunkType(t)$$

Fig. 8. Type-checking rules for Ptolemy. Rules for standard OO features are omitted.

The type checking of method and binding declarations within class c produces a type of the form OK in c, in which c can be considered an inherited attribute. Thus the rule (CHECK BINDING) works with such an inherited attribute c. It checks consistency between c's method m and the PCD. PCD types contain a return type c' and a type environment π, and all but the first formal parameter of the method m must be names defined in π with a matching type. The first formal parameter must be a thunk type that returns the same type, c', as the result type of the method.

Checking event type declarations involves checking that each type used is declared.

The type checking of PCDs involves their return type and the type environment that they make available [32, 35]. The return type and typing context of a named PCD are declared where the event type named is declared. For example, the `FEChange` PCD has `FElement` as its return type and the typing context that associates `changedFE` to the type `FElement`.

For a disjunction PCD, the return type is the least upper bound of the two PCDs' return types, and the typing context is the intersection of the two typing contexts. For each name I in the domain of both contexts, the type exposed for I is the least upper bound of the two types assigned to I by the two PCDs.

Expressions are type checked in the context of a local type environment Π, which gives the types of the surrounding method's formal parameters and declared local variables. Type checking of **`under`** and **`register`** is straightforward.

In an expression of the form **`proceed`**`(`e`)`, e must have a type of the form **`thunk`** c, which ensures that the value of e is a proceed closure. The type c is the return type of that proceed closure, and hence the type returned by **`proceed`**`(`e`)`.

In an **`event`** expression, the result type of the body expression, c', must be a subtype of the result type c declared by the event type, p. Furthermore, the lexical scope available (at e) must provide the context demanded by p.

The proof of soundness of Ptolemy's type system uses a standard preservation and progress argument [38]. The details are contained in our technical report [35].

2.4 Ptolemy's Compiler

We designed an extension of Java to have quantified, event types and implemented a compiler for this extension using the Eclipse's JDT core package [39]. Our prototype compiler [40] is backwards compliant; i.e., all valid Java code is valid Ptolemy code. It also generates standard Java byte-code. In the rest of the section, we describe the extensions to the Eclipse JDT Core we used to support quantified, event types.

We modified the scanner and parser of Eclipse JDT (contained in the package `org.eclipse.jdt.internal.compiler.parser`) to parse Ptolemy's new constructs (namely **`evtype`**, **`event`**, **`register`**, and bindings). Events were added as both expressions and statements, since Java makes this distinction. These modifications were fairly modular and did not require changing the existing structure of Eclipse's Java grammar; however, for automating the (extremely manual and error prone) parser building process of Eclipse some modifications to the type-hierarchy of the parser and its parser generation tool (*jikespg*) were made.

Eclipse's Java document object model (JDOM) was extended to include `EventTypeDeclaration` as a new `TypeDeclaration`, `EventStatement` as a new `Statement`, `EventExpression` and `RegisterExpression` as new subclasses of `Expression`, and `BindingDeclaration` as a new `TypeMemberDeclaration`.

Standard OO type checking rules are already implemented in Eclipse JDT. The semantic analysis is organized in a style similar to the composite design pattern [9], where both the composite and the leaf nodes provide uniform interface and the operation in the composite is implemented by recursively calling the operation on components. A visitor structure (`ASTVisitor`) is also provided, but the internal semantic analysis and code generation process does not use this structure. To add the type-checking rules for Ptolemy described in Section 2.3, we simply implemented them in the new AST nodes. The code generation for new AST nodes was also implemented similarly. These two steps also did not require modifications to implementation of other AST nodes.

Detailed description is beyond the scope of this paper, however, briefly the code generation proceeds as follows. Corresponding to an event type a set of classes and interfaces are generated that serve to model event frames, proceed closures, and event handlers. A closure object containing the body of event expression or statement is created as an inner class that replaces the original expression or statement. This inner class implements the interface that represents the event type at runtime and provides an implementation of the proceed method, which contains the original event's body. The replacement of the body requires a def-use analysis [41] with respect to its original environment and some name and reference mangling to propagate side-effects.

The class representing the event frame creates a chain of linked frames during registration that are parametrized with proceed closures during event invocation, as in the (EVENT) and (PROCEED-RUN) rules in Figure 6. Much of this is similar to the intuition discussed in Ptolemy's operational semantics in Section 2.2.

3 Comparisons with II and AO Langauges

The most perfect political community must be amongst those who are in the middle rank. – Aristotle, Politics

In this section we compare Ptolemy with II and AO languages. We start with an extended example that illuminates some differences between Ptolemy and AO languages.

3.1 An Extended Example in Ptolemy

In the extended example presented in this section, we use notations closer to a full-fledged language such as Java, such as **`if`** statements. Such constructs can be easily added to Ptolemy's core language.

The example shown in Figure 9 extends the example from Section 1. A set of classes are added to facilitate storing several figure elements in collections, e.g. as a linked list (`FEList`), as a set (`FESet`), and a hash table (`FEHashtable`). Furthermore, `Counter` implements the policy that whenever an `FElement` is added to the system a count must be incremented.

```
1 FElement evtype FEAdded {FElement addedFE;}

3 class FEList extends Object {
4  Node listhead;  /*head of linked list*/
5  FElement add(FElement addedFE) {
6   event FEAdded {
7    Node temp = listhead;
8    listhead = new Node();
9    listhead.data = addedFE;
10   listhead.next = temp; addedFE
11   }
12  }
13  FElement remove(FElement fe) { /*...*/ }
14  boolean contains(FElement fe) { /*...*/ }
15 }
16 class FESet extends FEList { /* ... */
17  FElement add(FElement addedFE) {
18   if(!this.contains(addedFE)) {
19    event FEAdded {
20     Node temp = listhead;
21     listhead = new Node();
22     listhead.data = addedFE;
23     listhead.next = temp; addedFE
24    }
25   } else { null }
26  }
27 }

29 class Counter extends Object {
30  Number count;
31  Counter init() {
32   register(this)
33  }
34  FElement increment(thunk FElement next,
35    FElement addedFE) {
36   this.count = this.count.plus(1);
37   proceed(next)
38  }
39  when FEAdded do increment
40 }

42 Counter u = new Counter().init();
43 /* ... */
```

Fig. 9. Figure Element Collections in Ptolemy

The notion of "adding an element" differs among the different types of collection. For example, calling `add` on a `FEList` always extends the list with the given element. However, calling `add` on a `FESet` only inserts the element if it is not already present, as shown on lines 16–20. Therefore, an AO-style syntactic method of selecting events such as "an `FElement` is being added" will need to distinguish which calls will actually add the element. In a language like AspectJ, one could use an **`if`** PCD. A PCD such as **`call`**`(* FESet.add(FElement fe)) && `**`this`**`(feset) && `**`if`**`(!feset.contains(fe))` would filter out undesired call events.

However, there are two issues with using such an **`if`** PCD. The first issue is that it exposes the internal implementation details of `FESet.add` (in particular that its representation does not allow duplicates). Second, such a PCD should only be used if the expression `feset.contains(fe)` does not have any side-effects. (Side-effects would usually be undesirable when used solely for filtering out undesired events.)

Other possibilities for handling such events include: (1) testing the condition in the handler body and (2) rewriting the code for `FESet.add` to make the body of the **`if`** a separate method call. The first has problems that are similar to those described above with using an **`if`** PCD. Rewriting the code to make a separate method call obscures the code in a way that may not be desirable and may cause maintenance problems, since nothing in the syntax would indicate why the body of the called method was not used inline. There may also be problems in passing and manipulating local variables appropriately in such a body turned into a method, at least in a language like Java or C# that uses call by value.

Such workarounds are also necessary in more more sophisticated AO languages such as LogicAJ [17]. These have PCDs that describe code structure, but that does not prevent undesirable exposure of internal implementation details, since the structure of the code is itself such a detail.

By contrast, Ptolemy easily handles this problem without exposing internal details of `FESet`'s `add` method, since that method simply indicates by when the event `FEAdded` occurs. In essence, Ptolemy's advantage is that it can explicitly announce the body of an **`if`** as an event. Doing so precisely communicates the event without the problems of using **`if`** PCDs or extra method calls described above.

3.2 Advance over Implicit Invocation Languages

Consider the II implementation of our drawing editor example (Figure 1). Compared to that implementation, in Ptolemy registration is more automated (see Figure 4), so programmers do not have to write code to register an observer for each separate event.

Ptolemy's registration also better separates concerns, since it does not require naming all classes that announce an event of interest. This is because events are not considered to be attributes of the classes that announce them. Thus, event handlers in Ptolemy need not be coupled with the concrete implementation of these subclasses. Furthermore, naming an event type, such as `FEChange`, in a PCD hides the details of event implementation and allows succinct quantification.

Ptolemy can also replace (or override) code for an event (like AO's "around advice"). Although similar functionality can be emulated, Ptolemy's automation significantly eases the programmer's task.

3.3 Advance over AO Languages

Some of the advantages of named event types would also be found in a language like AspectJ 5, which can advise code tagged with various Java 5 annotations. If one only advises code that has certain annotations, then join points become more explicit, and more like the explicitly identified events in Ptolemy. However, Java 5 cannot attach annotations to arbitrary statements or expressions, and in any case such annotations do not solve the problems described in the rest of this section.

Robust Quantified, Typed Events If instead of lexical PCDs Ptolemy's event expressions are used to announce events and PCDs are written in terms of these event names, innocuous changes in the code that implements the events will not change the meaning of the PCDs. For further analysis of robustness against such changes, let us compare the syntactic version of the PCD `target(fe) && call(FElement+.set*(..))` with Ptolemy's version in Figure 4. The syntactic approach to selecting events provides ease of use, i.e., by just writing a simple regular expression one can select events throughout the program. But this also leads to inadvertent selection of events: `set*` may select `setParent`, which perhaps does not change the state of a figure element. AO languages with sopisticated matching, based on program structure [17] or event history [20], have more possibilities for precise description of events, but can still inadvertently select unintended events. Ptolemy's quantified typed events do not have this problem.

Flexible Quantification The **event** expression in Ptolemy allows one to label any expression as an event expression and all such events can be selected by using the event type name in a PCD. Significant flexibility comes from giving developers the ability to decide what expressions constitute events and making them all available for quantification purposes. This largely solves the quantification failure problem [24]. The events that can be made available to handlers are no longer limited to interface elements, and the implementations of these events are not exposed to handlers. Handlers only rely on event type declarations. In contrast to implicitly announced events in AO languages, Ptolemy's **event** expression allows one to announce any expression as an event.

Flexible Access to Context Information The third problem that we considered in Section 1 was the difficulty of retrieving context information from a join point. Event types in Ptolemy solve this problem. To make the reflective information at the event available, a programmer only needs to define, in the lexical scope surrounding the event expression, values for the names declared in that event's type. For example, in Figure 4 in the `setX` method a block expression assigns **this** to `changedFE`. Note that this flexibility does not introduce additional coupling between events and handlers. Handlers are only aware of the context variable declaration `changedFE` made available by the event type `FEChange` and not of the concrete mapping to the variables available in the lexical scope of the event expression.

Uniform Access to Irregular Context Information AO join point models currently do not provide uniform access to irregular contextual information. But Ptolemy's event expressions allow uniform access to such context information. For example, in Figure 4, the event expression in the `setX` method and in the `makeEqual` method are given different bindings for the context variable `changedFE`, yet the handler `update` is able to access this context information uniformly using the event type name `changedFE`. The implementation details are also hidden from PCDs, which can uniformly access the context provided at the event (e.g., using the event type `changedFE`).

Concern and Obliviousness Both AO languages and Ptolemy have advantages for certain programming tasks. Consider first whether the concern needs to affect the code in which the events happen — the *base code* in AO terminology. "Spectator" concerns, like tracing, do not affect the base code's state [42, 36, 43]. Since spectators do not affect reasoning about the base code, explicit announcements in the base code give little benefit. Hence the determining factor is whether PCDs are easier to write lexically (in the AO style) or using explicitly named events (as in Ptolemy). For syntactically unrelated pieces of code, e.g., the locations of the event `FEAdded` in Figure 9, explicit announcement makes writing such PCDs more convenient. However, if the events occur in sections of the base code that are syntactically related (by a naming convention, placement in a common package, etc.), then lexical PCDs are preferable.

Besides the availability of uniform context (as described above) another property that affects how easy it is to write lexical PCDs is whether events in the base code are explicit at a module's interface, e.g., calls or executions of a public method. As pointed out by Aldrich [26] internal events should not be implicitly exported, hence explicit announcement should be used for such events to force negotiation about the commitments involved in having spectators rely on these events.

"Assistants" (i.e., non-spectators [42]) have handlers that affect the base code's state. Hence events handled by assistants are important for reasoning about the base code's state. With implicit announcement it is difficult to see these events and take them into account during reasoning. Furthermore, conclusions drawn about the base code will change depending on which assistants are added to the program. Thus we believe that events that are of concern to assistants should always be explicitly announced.

In conclusion, implicit announcement—obliviousness—is useful for spectator concerns when it is easy to write lexical PCDs. In other cases, Ptolemy's explicit event announcement and its event model are better.

4 Comparative Analysis with Related Work

"There is no other royal path which leads to geometry," said Euclid to Ptolemy I.

In this section, we compare Ptolemy with other mechanisms that address similar problems in AO language design. The other mechanisms we selected for analysis include Aspect-Aware Interfaces (AAIs) [11], Open Modules (OMs) [26], and Crosscut Programming Interfaces (XPIs) [24] [25]. Next section summarizes these ideas.

4.1 Overview of Related Ideas

Aspect Aware Interfaces (AAIs) [11] show dependencies between code and handlers. The whole program's configuration, which contains all classes and bindings (including PCDs) is first used to compute dependencies between events and handlers (called the "global step" [11]). The result of this global step is similar in some ways to code in Ptolemy, since one can look at an AAI and see where events may occur that will call handlers, and what handlers may be called for such events. However, whenever the program's bindings are changed, the global step must be repeated and an entirely new set of implicitly announced events might be handled, causing new dependencies. Ptolemy's event expressions do not declare what handlers are applicable for the event they explicitly announce, but the use of explicit announcement ensures that changing a program's bindings will not advise other (previously unanticipated) program points. AAIs also give no help with the problems discussed in Section 1.

Aldrich's Open Modules (OMs) proposal [26] is closely related to this work and has similar advantages. Like our work, OMs also allows a class developer to explicitly expose the sets of events that are announced. The implementations of these events remain hidden from PCDs and handlers. As a result, the impact of code changes within the class on PCDs is reduced. However, in OMs each explicitly exposed PCD has to be enumerated when binding handlers to sets of events (i.e., when writing advice). By contrast, Ptolemy's event types provide significantly simpler quantification. In Ptolemy, instead of enumerating the events of interest, one can use the event types for more convenient non-syntactic quantification to select join points. As with OMs, a programmer using Ptolemy's event types must systematically modify modules in a system that a given

Criteria	Description	AAIs	OMs	XPIs	Ptolemy
Abstraction	Supports abstraction?	Yes	Yes	Yes	Yes
Aspect/Base IH	Is information hiding supported for aspect / base?	Aspect	Base	Aspect + Base	Aspect + Base
Reasoning	What is the granularity of reasoning?	Join point	Module	XPI's Scope	Expression
Configuration	Requires complete system configuration?	Yes	No	No	No
Decoupling	Decouples aspects from base code?	No	Yes	Yes	Yes
Locality	Are interface definitions textually localized?	No	No	Yes	Yes
Stable	Is it stable against code changes?	Low	High	Medium	High
Pattern	Allows pattern-based quantification?	Yes	in module	in XPI's scope	No
Type	Allows quantification based on type hierarchy?	No	No	No	Yes
Scope	What is the scope of the interface?	Program	Module	User defined	User defined
Scope control	Has fine-grained control over scope?	No	No	No	Yes
Adaptation	Requires base code adaption / refactoring?	No	Yes	Yes	Yes
Oblivious	Is it purely oblivious?	No	No	No	No
Lexical hints	Provides lexical hints in a module?	Yes	Yes	No	Yes

Fig. 10. Results of comparative analysis

concern crosscuts to expose events that are to be advised, by using **`event`** expressions. For example, the module *Point* in Figure 4 had to be modified to expose events of type `FEChange`. However, unlike OMs, once modules have incorporated such **`event`** expressions, no awkward enumeration of explicitly exposed join points is necessary for quantification. Instead, one simply uses the event type `FEChange` in a PCD. Furthermore, in Ptolemy one can expose events that are internal to a module, such as the bodies of **`if`** statements (Figure 9, lines 17–20), which is not possible in OMs.

Sullivan *et al.* [24] proposed a methodology for aspect-oriented design based on design rules. The key idea is to establish a design rule interface that serves to decouple the base design and the aspect design. These design rules govern exposure of execution phenomena as join points, how they are exposed through the join point model of the given language, and constraints on behavior across join points (e.g. *provides* and *requires* conditions [25]). These design rule interfaces were later called crosscut programming interface (XPI) by Griswold *et al.* [25]. XPIs prescribe rules for join point exposure, but do not provide a compliance mechanism. Griswold *et al.* have shown that at least some design rules can be enforced automatically. In Ptolemy, enforcing design rules is equivalent to type checking of programs, because Ptolemy's event types automatically provide the interfaces needed to decouple different modules.

4.2 Criteria and Analysis Results

The criteria and the analysis results are summarized in Figure 10. The rest of this section presents our analysis in detail.

Abstraction, Information Hiding The first criterion is whether the approach supports abstraction. All four approaches support abstraction. AAIs abstract the advice that is being executed at the join point, while providing information about the advising structures in a specific system deployment scenario. Their automatically computed abstraction is useful for the developer of the base code in hiding the details of the aspects that may come to depend on the base code. OMs abstract the join point implementation by providing an explicitly declared pointcut as part of the module description. Their abstraction is useful for the aspect code and hides the details of the base code. XPIs provide an abstraction for a set of join points to the aspects, and an abstraction for the possible cumulative behavior of all advice constructs to the base program through their requires/provides clauses. Ptolemy provides an abstraction for a set of events to the handlers. It also provide a two-way abstraction for all context information exchanged between an event expression and the handler.

Modular Reasoning and the Role of the System Configuration All four approaches support different mechanisms for modular reasoning. AAIs are different from OMs, XPIs and Ptolemy in that they require that dependencies between base code and aspects be computed before modular reasoning can begin. This may preclude reasoning about a module, until all aspects and classes are known. OMs are geared towards supporting reasoning about a change inside a module without knowing about all aspects and classes present in the system. By ensuring that no aspects come to depend upon the changeable implementation details, the need to pre-compute all base-aspect dependencies is eliminated. XPIs are geared towards supporting reasoning about a change inside a scope. Ptolemy allows reasoning at the expression level; in particular, only event expressions require any special treatment compared with OO programs.

Locality This criterion evaluates whether the AO interface definitions are textually localized. AAIs are computed for each point where advice might apply, and thus are not localized. OMs are similar in that the interface of each module explicitly specifies the join points exposed by that module. In XPIs, the AO interface definitions are localized as an abstract aspect. In Ptolemy the event expressions are not localized but the type definition that serves as an interface to the handlers is localized.

Pattern-based Quantification, Scope, and Scope Control Mechanisms AAIs, OMs and XPIs all support pattern-based quantification. The difference lies in the scope of application of the pattern-based quantification techniques. The scope in the case of AAIs is generally the entire program, but can be limited to specific regions using lexical pointcut expressions such as **`within`** and **`withincode`**. In OMs, they are applicable to inside a module only if used to declare explicitly exposed pointcut and to the entire program if used to select interface elements of modules. XPIs have an explicit scope component that can serve to limit the effect of pattern-based quantification, which in turn is implemented using the **`within`** and **`withincode`** PCDs. In Ptolemy, one can only select program execution events that are declaratively identified. A much finer-grained scope control is available in the case of Ptolemy. In other approaches scope control depends on the language's expressiveness.

Base Code Adaptation and Obliviousness Obliviousness is a widely accepted tenet for aspect-oriented software development [16]. In an oblivious AO process, the designers and developers of base code need not be aware of, anticipate or design code to be advised by aspects. This criterion, although attractive, has been questioned by many [26, 42, 43, 25, 11, 44, 24]. All four approaches limit the notion of obliviousness to some extent. In Ptolemy adapting base code is necessary.

5 Other Related Ideas

advertise, annunciate, broadcast, declare, proclaim, promulgate, publish
– entry for "announce" in Roget's II

Not all aspect-oriented languages quantification is based on pattern matching of lexical names. For example, in LogicAJ [17] and similar languages such as LogicAJ2, Sally [45], quantification is based on lexical structures of program, in languages that support trace-based pointcuts [46], quantification is based on program traces. As mentioned before, such languages, although significantly expressive compared to the AspectJ-like languages that quantify based on names, also exhibit fragile pointcut problem. Compared to this entire class of such AO languages, which quantify based on pattern matching, Ptolemy's quantified event types in Ptolemy further decouple event handlers and the code that signals events and encapsulates the details of the signaller's code. However, upfront efforts will be required in Ptolemy to anticipate and announce events.

Explicitly labeling methods for use in quantification is not a new idea and has appeared previously in SetPoint [29] and Model-based Pointcuts [23]. In SetPoint explicitly placed annotations are used for quantification. In Model-based Pointcuts, explictly created models, which express the relationship between names in the model and the lexical structure, are used for quantification. Compared to these approaches, the novelty of our approach lies in: allowing arbitrary expressions to be announced as events, in providing explicitly announced events with types, in formalizing the language's sound, static type system, and in providing access to the context of event announcements. Compared to model-based pointcuts, our technique does not require a model construction step. Furthermore, keeping such model consistent with the code can be challenging.

Steimann and Pawlitzki have independently advanced ideas that are very similar [47]. Their language has event types and explicitly announced events that contain arbitrary statements. Their event types are similar to Ptolemy's. Their language is a modification of AspectJ, and has both implicit (AO style) and explicit announcement of events, whereas Ptolemy only has explicit announcement. In their language explicit announcement passes context positionally (as in a constructor call), whereas in Ptolemy context is passed by name matching. Their language is also somewhat similar to Open Modules in that the event types that are exported by a class must be declared by that class. They also have a prototype implementation, but do not formally present their language's semantics or type system.

Delegates in .NET languages such as C# and Java's `EventObject` class are also related to our approach. They are type-safe mechanism for implementing call back functions that can also be used to abstract event declaration code; however, these mechanisms do not provide the quantification feature of Ptolemy's PCDs.

Another related area is mediator-based design styles [6]. In this design style modules tell mediators about event declarations and announcements. Other modules can register with mediators to have their methods invoked by event announcements. An invocation relation is thus created without introducing name dependencies. In our approach, event types play the role of mediators. However, in Ptolemy, one can also use event types for quantification, which simplifies registration and binding. By contrast, in mediator based designs a developer has to resort to explicit and possibly error-prone enumerations to during to register handlers for events.

Consider a language with closures and the ability to reflectively get the run time context of a statement or expression. In such language, one could achieve the same effect as Ptolemy's quantified event types by declaring classes to represent events, announcing events by creating a closure after reflectively accessing the event body's run time context and then looping over a set of registered handler methods, passing each a closure (that it could proceed to). Ptolemy provides three advantages over this emulation:

– **Static typechecking** of bindings, which ensures that PCDs only associate handlers with events that provide the necessary context.
– A considerable amount of **automation**. Ptolemy's **`register`**, **`event`**, and **`proceed`** expressions hide the details of registration, announcement, and running handlers. Furthermore PCDs provide quantification, which is not easy to emulate.
– **Improved compiler optimizations**. Since Ptolemy controls the details of how registration, announcement, and running handlers are implemented, there is more potential for optimization then when these features are emulated.

6 Future Work and Conclusions

Onward and upward. — Abraham Lincoln

We designed Ptolemy to be a small core language, in order to clearly communicate its novel ability to announce arbitrary expressions as events and its use of quantified, event types, and in order to avoid complications in its theory. However, this means that many practical and useful extensions had to be omitted from the language. The most important future work in the area of Ptolemy's semantics is subtyping of event types and investigating the possible advantages of positional context exposure (instead of Ptolemy's name-based context exposure). We have already extended Ptolemy's operational semantics to include control flow ("cflow") PCDs [35], which are not discussed in this paper due to lack of space. It would also be interesting to combine Ptolemy's type system with an effect system, to limit the potential side effects of handler methods [32, 36]. This might allow more efficient reasoning. One could also imagine combining specifications of handler methods into code at **`event`** expressions, thus allowing verification of code that uses event types to be more efficient and maintainable than directly reasoning about the compiled code's semantics. In general, a detailed investigation of specification and verification issues for Ptolemy would be very interesting.

In conclusion, the main contribution of this work is the design of a language, Ptolemy, with quantified, typed events. In addition to a precise operational semantics and formal definition of Ptolemy's type system (see our technical report for a soundness proof [35]), we have carefully examined how Ptolemy fits in the design space of languages that promote separation of concerns. The main new feature of Ptolemy is event types, which contain information about the names and types of exposed context. In Ptolemy's new event model, events are explicitly announced by event expressions, which declaratively identify the type of event being announced. These event types are used in PCDs to associate handlers with sets of events. Such PCDs are robust against code changes, since they are only affected by changes to event expressions. The event types used in PCDs make it easier for handlers to uniformly access reflective information about the events without breaking encapsulation. Ptolemy has been implemented as a compiler, and an implementation is available for free download [40].

Ptolemy's ability to announce any expression as an event, which permits one to expose internal states in a principled way, promises real value. For example, this would help the integration of components when hidden internal states and transitions must be accessed in order to achieve proper composition.

References

1. Dingel, J., Garlan, D., Jha, S., Notkin, D.: Reasoning about implicit invocation. SIGSOFT Software Engineering Notes **23**(6) (November 1998) 209–21
2. Garlan, D., Notkin, D.: Formalizing design spaces: Implicit invocation mechanisms. In: VDM '91. (1991) 31–44
3. Luckham, D.C., Vera, J.: An event-based architecture definition language. IEEE Trans. Softw. Eng. **21**(9) (1995) 717–734
4. Notkin, D., Garlan, D., Griswold, W.G., Sullivan, K.J.: Adding implicit invocation to languages: Three approaches. In: JSSST International Symposium on Object Technologies for Advanced Software. (1993) 489–510
5. Reiss, S.P.: Connecting tools using message passing in the Field environment. IEEE Softw. **7**(4) (1990) 57–66
6. Sullivan, K.J., Notkin, D.: Reconciling environment integration and software evolution. ACM Transactions on Software Engineering and Methodology **1**(3) (July 1992) 229–68
7. Kiczales, G., Lamping, J., Menhdhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: ECOOP 97, Finland (Jun 1997) 220–242
8. Tarr, P., Ossher, H., Harrison, W., Sutton, S.: N degrees of separation: Multi-dimensional separation of concerns. In: ICSE '99
9. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design patterns: elements of reusable object-oriented software. Addison-Wesley Publishing Co., Inc. (1995)
10. G. Kiczales et al.: An overview of AspectJ. In: ECOOP '01, Jun 2001, 327–353
11. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: ECOOP 2005. 195–213
12. Filman, R.E., Elrad, T., Clarke, S., Akşit, M., eds.: Aspect-Oriented Software Development. Addison-Wesley, Boston (2005)
13. Elrad, T., Filman, R.E., Bader, A.: Aspect-oriented programming: Introduction. Commun. ACM **44**(10) (2001) 29–32
14. Laddad, R.: AspectJ in Action: Practical Aspect-Oriented Programming. Manning (2003)
15. AspectJ Team: The AspectJ programming guide. Version 1.5.3. Available from `http://eclipse.org/aspectj` (2006)
16. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns (OOPSLA '00). (Oct. 2000)
17. Rho, T., Kniesl, G., Appeltauer, M.: Fine-grained generic aspects. In: FOAL'06
18. Eichberg, M., Mezini, M., Ostermann, K.: Pointcuts as functional queries. In: APLAS 04. 366–381
19. Douence, R., Fradet, P., Sudholt, M.: Trace-based aspects. Aspect-oriented Software Development 141–150
20. Stolz, V., Bodden, E.: Temporal assertions using AspectJ. In: Fifth Workshop on Runtime Verification (RV '05). (2005)

21. Stoerzer, M., Graf, J.: Using pointcut delta analysis to support evolution of aspect-oriented software. In: ICSM '05. 653–656
22. Tourwé, T., Brichau, J., Gybels, K.: On the existence of the AOSD-evolution paradox. In: SPLAT '03. (March 2003)
23. Kellens, A., Mens, K., Brichau, J., Gybels, K.: Managing the evolution of aspect-oriented software with model-based pointcuts. In: ECOOP '06. 501 – 525
24. Sullivan, K.J., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: Information hiding interfaces for aspect-oriented design. In: ESEC/FSE 2005. 166–175
25. W. G. Griswold et al.: Modular software design with crosscutting interfaces. IEEE Software (Jan/Feb 2006)
26. Aldrich, J.: Open Modules: Modular reasoning about advice. In: ECOOP '05. 144–168
27. Harbulot, B., Gurd, J.R.: A join point for loops in AspectJ. In: AOSD 06. (2006) 63–74
28. Rajan, H., Sullivan, K.J.: Aspect language features for concern coverage profiling. In: AOSD 2005. 181–191
29. Altman, R., Cyment, A., Kicillof, N.: On the need for Setpoints. In: European Interactive Workshop on Aspects in Software. (2005)
30. Rajan, H., Sullivan, K.J.: Classpects: unifying aspect- and object-oriented language design. In: ICSE 2005. 59–68
31. Mezini, M., Ostermann, K.: Conquering aspects with Caesar. In: AOSD 03. 90–99
32. Clifton, C.: A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University (Jul 2005)
33. Clifton, C., Leavens, G.T.: MiniMAO$_1$: Investigating the semantics of proceed. Science of Computer Programming **63**(3) (2006) 321–374
34. Rajan, H., Sullivan, K.J.: Eos: instance-level aspects for integrated system design. In: ESEC/FSE 2003. 297–306
35. Rajan, H., Leavens, G.T.: Quantified, typed events for improved separation of concerns. Technical Report 07-14c, Iowa State University, Dept. of Computer Sc. (October 2007)
36. Clifton, C., Leavens, G.T., Noble, J.: MAO: Ownership and effects for more effective reasoning about aspects. In: ECOOP '07. 451–475
37. Flatt, M., Krishnamurthi, S., Felleisen, M.: A programmer's reduction semantics for classes and mixins. In: Formal Syntax and Semantics of Java. Springer-Verlag (1999) 241–269
38. Wright, A.K., Felleisen, M.: A syntactic approach to type soundness. Information and Computation **115**(1) (Nov 1994) 38–94
39. Eclipse Foundation: Eclipse website. `http://www.eclipse.org/`
40. Rajan, H., Leavens, G.T.: Ptolemy website. `http://www.cs.iastate.edu/~ptolemy/` (2007)
41. Nielson, F., Nielson, H.R., Hankin, C.: Principles of Program Analysis. Second printing edn. Springer-Verlag (2005)
42. Clifton, C., Leavens, G.: Observers and assistants: A proposal for modular aspect-oriented reasoning. In: FOAL 02. 33–44
43. Dantas, D.S., Walker, D.: Harmless advice. In: POPL 06. (2006) 383–396
44. Steimann, F.: The paradoxical success of aspect-oriented programming. In: OOPSLA '06. (October 2006) 481–497
45. Hanenberg, S., Unland, R.: Parametric introductions. In: AOSD '03. 80–89
46. Douence, R., Motelet, O., Südholt, M.: A formal definition of crosscuts. In: REFLECTION '01. 170–186
47. Steimann, F., Pawlitzki, T.: Types and modularity for implicit invocation with implicit announcement. Obtained from the first author. (August 2007)