Instance-level Quantified, Typed Events for Integrated System Design

Mehdi Bagherzadeh, Robert Dyer, Yuheng Long, and Hridesh Rajan

TR #08-15
Initial Submission: Dec 23, 2008.

Keywords: component integration, implicit-invocation languages, aspect-oriented programming languages, quantification, pointcut, join point, context exposure, type checking, event types, event expressions, instance-level aspects.
CR Categories:
D.2.10 [*Software Engineering*] Design
D.1.5 [*Programming Techniques*] Object-Oriented Programming
D.2.2 [*Design Tools and Techniques*] Modules and interfaces,Object-oriented design methods
D.2.3 [*Coding Tools and Techniques*] Object-Oriented Programming
D.3.3 [*Programming Languages*] Language Constructs and Features - Control structures

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

Instance-level Quantified, Typed Events for Improved Separation of Integration Concerns

Mehdi Bagherzadeh, Robert Dyer, Yuheng Long, and Hridesh Rajan

Department of Computer Science, Iowa State University
226 Atanasoff Hall Ames, IA, 50011, USA
{mbagherz,rdyer,csgzlong,hridesh@cs.iastate.edu}

Abstract. Integrated systems are those where components must behave together in order to fulfill overall requirements. In such systems, modularization of integration relationships is important for enabling separate component compilation, testing, and debugging, and for enhanced reuse. Existing languages and approaches for modularizing integration relationships work, but do not solve all problems. In particular, they either do not completely decouple components or require workarounds, which at a minimum incurs design and performance overheads. In this work, we discuss instance-level quantified, typed events, which solve all of these problems. The technical contributions include: the design, semantics, and type system of instance-level quantified, typed events and a proof of its soundness. A formalized semantics is new to this paper, as there have been no previous formalizations of language features that aim to modularize separation of integration relationships. To demonstrate the feasibility of our language design, we have implemented this design in an interpreter. To provide an initial assessment of the language's benefits, we have implemented canonical examples in the literature. Our initial assessments show that instance-level quantified, typed events improve the separation of integration concerns over previous language design proposals.

1 Introduction

Integrated systems [24, 29] are a broad class of software systems in which logically unrelated objects, such as compilers, editors, debuggers, or any other kind of discrete components must interact dynamically to meet system-level requirements (e.g., the editor must automatically open the right file and scroll to the right line when the debugger encounters a breakpoint) [16]. In the context of these systems, the modularization of *behavioral relationships* is an important problem [29, 20, 22]. These relationships coordinate the control, actions, and states of subsets of system components to satisfy overall system requirements; thus, they are also referred to as *integration concerns* [16, 29].

Previous works have investigated the separation of integration concerns through instance-level advising e.g. the work of Pearce and Noble on relationship aspects [15], the work of Rajan and Sullivan on instance-level aspects in Eos [20, 22], and the work of Sakurai *et al.* on association aspects [26]. Others, like the work of Taner on expressive scoping of deployed aspects [5] and the work of Hoffman and Eugster [10], have suggested changing the advising model in Aspect-Oriented (AO) programming. Work on formalizing AO features has been done by Wand et al. [31], Clifton and Leavens [3], and others but to the best of our knowledge, no formalization has addressed instance-level features that require non-trivial extension.

In this paper we introduce and give a formal definition for instance-level event type support in Ptolemy [19], by adding instance-level event association to it. Ptolemy is a language with quantified, typed events that solves problems with previous Implicit Invocation (II) and Aspect-Oriented (AO) languages such as the lack of quantification in II [19] and fragile pointcuts in AO [19, 27, 30]. Ptolemy however has no support for instance-level event announcements and instead requires subjects to notify all registered instances of the observer when a specific event is announced. Systems dealing with separation of integration concerns require the ability to maintain these event relationships at a much finer granularity. Our solution is to add a new construct to Ptolemy, called *associate*, which allows creating such instance-level relationships. We give a formal semantics for our new language, called Ptolemy-I, and show several examples implemented in it. The examples are implemented using an interpreter we developed for Ptolemy-I, which shows the feasibility of instance-level type event support in Ptolemy-I.

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

- a language design that improves the separation of integration concerns [16, 29];
- a formalization of instance-level advising of events that plays a key role in the separation of integration concerns;
- a demonstration of the benefits of the language design via canonical examples presented elsewhere [16, 29]; and,
- an interpreter for the instance-level quantified, typed events as a proof of concept.

In the following section, we present insights into the problem through a motivating example. Section 3 shows how our language design solves the presented problem and talks about our language design. We then give detailed semantics of the language in Section 4 and a type system in Section 5. We evaluate our language design with a large canonical example implemented in our interpreter in Section 7. Finally, we discuss related work in Section 8 and conclude in Section 9.

2 Motivation

In this section, we motivate the need for instance-level quantified, typed events using an example borrowed from Rajan and Sullivan [20]. This system consists of a collection of models. A model provides the ability to change its state. A system-wide requirement is to keep the states of certain models *consistent* with each other, e.g. if model `m1` and `m2` are required to be consistent, whenever the state of `m1` changes, `m2` needs to be updated and vice-versa. This requirement is a representative of what Sullivan *et al.* have called behavioral relationships [29] in that control, actions and states of subsets of system components need to be coordinated to satisfy overall system requirements. The challenge is to enable modular representation of the *consistency* relationship.

In an object-oriented (OO) implementation of such a system, a model would be designed as an instance of an object-oriented class `Model` as shown in Figure 1 (lines 4–10). This class provides the ability to change the state of the model, which for simplicity is represented here as the method `Set` (lines 6–9). Other mutators are also conceivable.

The simplest implementation of the consistency requirement would be to keep a list of model instances as a field of the class `Model` and implement the consistency logic in the class `Model` itself. This would, however, couple the implementation of the consistency requirement with that of the model preventing their independent evolution and reuse. OO design patterns such as the observer pattern, as well as aspects [12], can be utilized for improved separation of consistency concern, but as Rajan and Sullivan [20, 23], Sakurai *et al.* [26], and Pearce and Noble [15] have shown, these do not fully modularize the integration relationships. These authors have proposed AO features in the style of AspectJ-like languages [12], e.g. instance-level aspects [20], association aspects [26], and relationship aspects [15], to further improve the modularization of such integration requirements. However, AspectJ-like AO languages have other problems in orthogonal dimensions, e.g. fragility of regular expression-based pointcuts [19, 27, 30].

In a previous work, Rajan and Leavens [19] proposed Ptolemy, a language design based on quantified, typed events, which solves the problems with AO languages and maintains similar benefits. To understand whether this language design enables improved separation of integration concerns, we implemented the consistency requirement using Ptolemy as illustrated in Figure 1 (lines 24–47, ignoring the grey code for now).

```
 1 Model evtype SetEvent {
 2   Model model;
 3 }
 4 class Model {
 5   bool value = false;
 6   Model Set() {
 7     Model model = this;
 8     event SetEvent { value = true; this }
 9   }
10 }
11 class Main {
12   Main main() {
13     Model m1 = new Model();
14     Model m2 = new Model();
15     Model m3 = new Model();
16     Model m4 = new Model();
17     Consistency c1 = new Consistency();
18     c1.AddMap(m1,m2); c1.AddMap(m3,m4);
19     m1.Set();
20     m3.Set();
21     this
22   }
23 }
```

```
24 class Consistency {
25   HashMap<Model, bool> busy = new ..;
26   HashMap<Model, Vector<Model>> map = new ..;
27   Consistency () { register(this) }
28   Consistency AddMap (Model m1, Model m2) {
29     /* initialize busy/map for m1 and m2 */
30     ..
31     map.get(m1).addElement(m2);
32     map.get(m2).addElement(m1);
33     this
34   }
35   Model HandleSetEvent (SetEvent inner) {
36     if (busy.containsKey(inner.model)) {
37       if (!busy.get(inner.model)) {
38         busy.put(inner.model, true);
39         for (Model m : map.get(inner.model))
40           if (!busy.get(m)) m.Set ();
41         busy.put(inner.model, false);
42       }
43     }
44     invoke(inner)
45   }
46   when SetEvent do HandleSetEvent;
47 }
```

Fig. 1. Model Example Implementation Using Ptolemy [19] (based on [20])

Ptolemy allows programmers to declare event types as shown in lines 1-3. These *event types* describe context information available, e.g. the event type `SetEvent` declares that the changing model instance is available as context. A programmer can name event types for declarative event announcement, e.g. on line 8 an *event expression* is used to announce the event `SetEvent`. Other classes may specify code to execute when events of a certain type occur in the application. For example for the consistency requirement, announcement of events of type "`SetEvent`" suggest that some model has changed and thus some other model may need to be updated. This is expressed on line 46 via a *binding declaration*, which specifies that the method `HandleSetEvent` should execute for events of type "`SetEvent`" and a *register expression* on line 27 that specifies the receiver object for the method `HandleSetEvent`. Note that the classes that announce such events need not be mentioned, which achieves the similar effect as a pointcut declaration in an AO language.

As a result of this event-related code, whenever a models' state changes by running the method `Set`, the code in the method `HandleSetEvent` is run, which implements the logic for consistency. The key advantage is that the implementation of the consistency requirement remains separate from that of the model and thus both can evolve independently, while depending on the interface provided by the event type `SetEvent`.

This implementation has one problem, however. In Ptolemy, there is no language-level mechanism to model the integration relationships between model instances. Instead, and like AspectJ-like AO languages, the method `HandleSetEvent` is run when any model instance announces "`SetEvent`". The implementation for consistency in Figure 1 implements a workaround to model integration relationships between instances (shaded code). First, code is added to maintain a hashmap containing models in the consistency relationship. Second, checks are added in the method `HandleSetEvent` to determine whether the model instance announcing `SetEvent` is participating in any relationships. The busy flag (line 25) must also be maintained (lines 36–38 and 40–41) for each model instance to prevent infinite loops.

This workaround has both design and performance costs. First, it unnecessarily complicates the implementation of the consistency requirement. Second, it creates a conceptual gap between the design view and the runtime view of the system. At the design-level, instances of models are integrated using instances of consistency relationships. To understand this system's behavior at runtime, e.g. for debugging purposes, a developers must create a mental map from each consistency relationship to the consistency pair inside the `map` field of the class `Consistency`. The performance costs are in unnecessary invocation of the method `HandleSetEvent`. Each model instance must pay the price of any model instance participating in an integration relationship. Such costs will grow with the number of model instances in the system and the number of integration relationships [20]. For a widely used class, e.g. a `List` of which several instances may exist in a system such performance costs may be prohibitive. In the next section, we discuss instance-level quantified, typed events which solves these problems.

3 Instance-Level Quantified, Typed Events

In this section, we describe Ptolemy-I's design. Ptolemy-I extends Ptolemy's design [19], thus it also contains features inspired from implicit invocation languages such as Rapide [13], and AO languages such as AspectJ [12], Eos [22] and Caesar [14]. Ptolemy-I features new mechanisms for relating object instances announcing events and handling events. In the rest of this section, we first describe the language design. We then revisit the `Model` example discussed in the previous section.

We have prototyped these features in an interpreter, which serves as a proof of concept for the language design. This interpreter is implemented in Scheme and strictly follows the semantics of Ptolemy-I described in Section 4, further increasing our confidence in the correctness of the language semantics. We have implemented and tested some canonical examples like the the `Model` example used in Section 2 and this section and the `Graph System` discussed in Section 7 using this interpreter.

3.1 Language Design

Ptolemy-I's abstract syntax is shown in Figure 2 and illustrated in Figure 3. In the spirit of Featherweight Java [11], Classic Java [6], and MiniMAO$_1$ [4], we have deliberately kept the core language small. It has classes, objects, inheritance, and sub-typing, but it does not have **`super`**, interfaces, exception handling, built-in value types, privacy modifiers, or abstract methods. A program may contain an arbitrary number of declarations followed by an expression that is treated as the entry point of the program. A declaration may be a class declaration or an event type declaration. A class declaration **`class`** may extend only one other class and may contain an arbitrary number of fields, methods, and bindings. An event type declaration **`evtype`** contains a return type c, a name p, and an arbitrary number of context variables. The field and method declarations have standard syntax. A binding declaration $binding$ is the keyword **`when`** followed by an event type name p followed by the keyword **`do`** followed by the name of the handler method m. The expressions contain standard OO expressions as well as Ptolemy-I specific expressions **`register`**, **`event`**, **`invoke`**, and **`associate`**.

```
prog    ::= decl* e
decl    ::= class c extends d { field* meth* binding* }
          | c evtype p { form* }
field   ::= c f;
meth    ::= t m (form*) { e }
t       ::= c | thunk c
binding ::= when p do m
form    ::= t var,      where var≠this
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 } | invoke( e ) | associate( e,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. 2. Abstract Syntax of Ptolemy-I, Based on Ptolemy [19]

Informally, an **`event`** expression (**`event`** `p { }`) is used for signaling an event of type p. The **`invoke`** expression is used for running the next handler method in the list of registered handler methods and the original event body. The **`register`** expression is used for putting a handler method in the list of registered handler methods. The intended semantics for these expressions are similar to that in Ptolemy [19].

The **`associate`** expression is new to Ptolemy-I. It serves to relate individual instances of subjects and observers. In **`associate`**(e, e'), e and e' are evaluated to observer and subject objects respectively, with *loc* and *loc*$'$ as locations. Then observer *loc* is associated to receive notification whenever subject *loc*$'$ announces any event p. Finally, the **`associate`** expression returns *loc*. To distinguish between subject-observer instances associated through **`register`** and **`associate`** expressions, a list containing binding records is used in the semantics to track subject-observer relationships.

3.2 Consistency Revisited

As discussed previously, the implementation of the consistency relationship using Ptolemy had design and performance overheads. This was primarily due to the workarounds necessary to simulate integration relationships between model instances. To solve this problem, Ptolemy-I introduces the notion of instance-level typed events, which allows individual instances of subjects and observers to be related with each other through the **associate** expression. Figure 3 presents an alternative implementation of the consistency relationship using instance-level quantified, typed events.

```
11 class Main {
12     ...
17     Consistency c1 = new Consistency();
18     Consistency c2 = new Consistency();
19     c1.AddMap(m1,m2); c2.AddMap(m3,m4);
20     ...
23 }
24 class Consistency {
25   bool busy = false;
26   Model m1, m2;
27   Consistency AddMap (Model m1, Model m2) {
28     associate(this, m1); this.m1 = m1;
29     associate(this, m2); this.m2 = m2;
30     this
31   }
32   Model HandleSetEvent (SetEvent inner) {
33     if (!busy) {
34       busy = true;
35       if (inner.model == m1) m2.Set();
36       else m1.Set();
37       busy = false;
38     }
39     invoke(inner)
40   }
41   when SetEvent do HandleSetEvent;
42 }
```

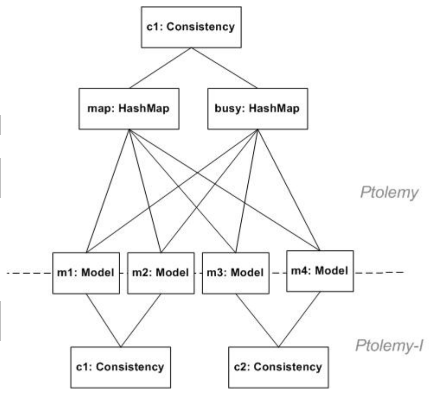

Fig. 3. Model Example Implementation In Ptolemy-I (**class** Model, **evtype** SetEvent and the remainder of **class** Main are the same as in Figure 1)

In this implementation, two example usages of the **associate** expression appear (lines 28–29). These associate Model instances m1 and m2 with the observer instance **this**. The effect of evaluating these two expressions is that the method HandleSetEvent is run only when the event SetEvent is announced in the method Set (Figure 1) and the currently executing instance is either m1 or m2.

We pointed out two design problems and a performance problem with the previously presented workaround in Section 2. From Figure 3 it is clear that Ptolemy-I's implementation is significantly simpler compared to the workaround presented previously. The simplification is primarily because now we do not need to keep track of model instances in an integration relationship using a hashmap. Rather, each instance of the class Consistency keeps track of the model instances that it relates. This leads to the second improvement over the previous design. The instances of consistency relationships in the design are now modeled using instances of the class Consistency at runtime (as can be seen in the diagram in Figure 3), which simplifies the conceptual gap between the static and dynamic representation of the program. Last but not least, the method HandleSetEvent is run only when necessary, thus avoiding the performance overheads due to unnecessary invocations. Instance-level quantified, typed events thus show to improve the modularization of integration concerns, while avoiding the need for workarounds with design and performance costs. Ptolemy-I thus brings the advantage of quantified, typed events to the design and implementation of integrated systems, which are shown to be an important class of software systems [24, 29].

With the language design proposal in place, we now turn to the semantics and the type system, which is also a contribution of this work over related ideas [21, 15, 26].

4 Semantics of Instance-level Quantified, Typed Events

This section defines a small-step operational semantics for Ptolemy-I. The technical description of the semantics follows the previous work of Rajan and Leavens [19], Clifton *et al.* [2, 4], and Flatt *et al.* [6]. As in the previous work [19, 2, 4, 6], a program's declarations are simply formed into a fixed list, which is used in the semantics of expressions. The small steps of the operational semantics thus gives a semantics of programs by giving a semantics of expressions [19]. This semantics relies on four expressions that are not part of Ptolemy-I's surface syntax to record final or intermediate states of the computation. 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 [19].

The small steps in the semantics are defined as transitions from one configuration to another. Figure 4 defines these configurations. A configuration is defined as the current expression e, stack of environments ρ, store μ and an ordered list ψ of binding records. The key challenge in the semantics and thus its novelty was in the integration of the semantics for the **`register`** expression and the **`associate`** expression that preserves the order of registered observers. We solve this problem by modeling a binding record as a variant record type with two variants: an associate record θ or a register record ϑ.

Stacks are an ordered list of frames, each frame recording the static environment and some other information. The type environments Π are only used in the type soundness proof in Section 6.

There are two types of frames. Lexical frames **`lexframe`** record an environment σ that maps identifiers to values. Event frames **`evframe`** are similar, but also record the event type name p being run. Store-able values are object records or event closures. In ordered list ψ, associate binding record θ is a tuple of the form $[loc_o, loc_s]$ where loc_o and loc_s are locations of observer instance and the subject instance, while register binding record ϑ only contains the location of observer instance ($[loc_o]$).

Values can be **`null`** or a location (loc). Object records consist of the name of the class and its fields stored in μ. An event closure **`eClosure`** is a list of handler records H, an event body e and its associated variable and typing environments. And finally each handler record h contains the location of receiver object, the handler method to be called on receiver object and an environment assembled from needed parameters.

In operational semantics, an evaluation context ($\mathbb{E}$) takes care of the operational semantics congruence rules and the order in which expressions are reduced [32].

The rules for the new expressions are given in Figure 5. For valid store locations loc and loc', the evaluation rule (ASSOCIATE) changes the configuration such that a new associate binding record θ of the form $[loc, loc']$ is added to the front of the list ψ. The evaluation rule (REGISTER) is similar with the subtle difference that here, a register binding record ϑ of form $[loc]$, containing only the location of the observer object, is

Added Syntax:

$e \quad ::= \ loc \mid \textbf{under}\ e \mid \texttt{NullPointerException} \mid \texttt{ClassCastException}$

where $loc \in \mathcal{L}$, a set of locations

Domains:

Γ	$::= \langle e, \rho, \mu, \psi \rangle$		"Configurations"
ρ	$::= \gamma + \rho \mid \bullet$		"Stack"
γ	$::= \textbf{lexframe}\ \sigma\ \Pi \mid \textbf{evframe}\ p\ \sigma\ \Pi$		"Frames"
σ	$::= \{var_k : v_k\}_{k \in K},$	**where** K is finite, $K \subseteq I$	"Environments"
μ	$::= \{loc_k \mapsto sv_k\}_{k \in K},$	**where** K is finite	"Stores"
v	$::= \textbf{null} \mid loc$		"Values"
sv	$::= o \mid ec$		"Storable Values"
o	$::= [c.F]$		"Object Records"
F	$::= \{f_k \mapsto v_k\}_{k \in K},$	**where** K is finite	"Field Maps"
ec	$::= \textbf{eClosure}(H)(e, \rho, \Pi)$		"Event Closure"
H	$::= h + H \mid \bullet$		"Handler Records List"
h	$::= \langle loc, m, \rho' \rangle$		"Handler Record"
ψ	$::= \beta + \Psi \mid \bullet$		"Subscriber List"
β	$::= \theta \mid \vartheta$		"Binding Records"
θ	$::= [loc_o, loc_s]$		"Associate Binding Record"
ϑ	$::= [loc_o]$		"Register Binding Record"

Type Attributes:

$\theta ::= \text{OK} \mid \text{OK in } c \mid \textbf{var}\ t \mid \textbf{exp}\ t$	"Type Attributes"
$\Pi ::= \{I : \theta_I\}_{I \in K},$ **where** K is finite, $K \subseteq (\mathcal{L} \cup \{\textbf{this}\} \cup \mathcal{V})$	"Type Environments"

Evaluation Relation: $\hookrightarrow : \Gamma \rightarrow \Gamma$

Evaluation Context:

$$\begin{aligned} \mathbb{E} \ ::= \ & - \mid \mathbb{E}.m(e\ldots) \mid v.m(v\ldots\mathbb{E}\,e\ldots) \mid \textbf{cast}\ t\ \mathbb{E} \mid \mathbb{E}.f \mid \mathbb{E}.f\text{=}e \\ \mid\ & v.f\text{=}\mathbb{E} \mid t\ var\text{=}\mathbb{E};\ e \mid \mathbb{E};\ e \mid \textbf{under}\ \mathbb{E} \mid \textbf{invoke}(\mathbb{E}) \\ \mid\ & \textbf{associate}(\mathbb{E}, e) \mid \textbf{associate}(v, \mathbb{E}) \mid \textbf{register}(\mathbb{E}) \end{aligned}$$

Fig. 4. Ptolemy-I Configuration, Domains and Evaluation Context based on [2, 4, 6, 19]

created and added to the front of the list ψ. The semantics of these two expressions ensure that observers are added to the list ψ in the same order as the execution of **associate** and **register** expressions.

The evaluation rule (EVENT) first extracts the event declaration from the program's declarations CT. Then using auxiliary function $hbind$ creates a list H of handler records h, where each record h contains the receiver object location (loc), handler method name (m) and an environment (ρ') that has the method call arguments for the handler method. And finally places this list inside the **invoke** expression, which starts running the handlers.

The auxiliary function $hbind$ in Figure 6 uses the subject instance location, stack and store plus the list of register/associate binding records ψ to produce a list of handler records that are applicable for the event announced in the current state. When called by the (EVENT) rule, $hbind$ has a new **evframe** on top of the stack containing the current event representation.

The rule (INVOKE) extracts the event closure from the store, goes through the list of event handler records, and runs handler methods one by one after assembling the needed environment. $\textbf{eClosure}(H)\ (e, \sigma, \Pi)$ contains an ordered list of handler records H, event body e, environment variable σ binding event context variables and their values and typing environment (Π) [19]. The event closure is run by **invoke**. The handler method m is looked up in CT using auxiliary function $findMeth(c, m)$ as shown in

Evaluation relation: $\hookrightarrow : \Gamma \rightarrow \Gamma$

(ASSOCIATE)

$$\dfrac{loc, loc' \in dom(\mu) \qquad \theta = [loc, loc'] \qquad \psi' = \theta + \psi}{\begin{array}{c}\langle \mathbb{E}[\mathtt{associate}(loc, loc')], \rho, \mu, \psi \rangle \\ \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu, \psi' \rangle\end{array}}$$

(REGISTER)

$$\dfrac{loc \in dom(\mu) \qquad \vartheta = [loc] \qquad \psi' = \vartheta + \psi}{\begin{array}{c}\langle \mathbb{E}[\mathtt{register}(loc)], \rho, \mu, \psi \rangle \\ \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu, \psi' \rangle\end{array}}$$

(EVENT)

$$\dfrac{\begin{array}{c}(c\ \mathtt{evtype}\ p\{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT \\ \sigma = envOf(\gamma) \qquad \sigma' = \{var_i \mapsto v_i \mid v_i = \sigma(var_i)\} \qquad loc \notin dom(\mu) \qquad loc' = \sigma(\mathtt{this}) \\ H = hbind(loc', \gamma' + \gamma + \rho, \mu, \psi) \qquad \Pi' = \{var_i : \mathtt{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{loc : \mathtt{var}\ (\mathtt{thunk}\ c)\} \\ \gamma' = \mathtt{evframe}\ p\ \sigma'\ \Pi' \qquad \mu' = \mu \oplus (loc \mapsto \mathtt{eClosure}(H)\ (e, \sigma, \Pi))\end{array}}{\langle \mathbb{E}[\mathtt{event}\ p\ \{e\}], \gamma + \rho, \mu, \psi \rangle \qquad \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ (\mathtt{invoke}(loc))], \gamma' + \gamma + \rho, \mu', \psi \rangle}$$

(INVOKE)

$$\dfrac{\begin{array}{c}\mathtt{eClosure}((\langle loc', m, \sigma' \rangle + H))\ (e, \sigma, \Pi) = \mu(loc) \\ [c.F] = \mu(loc') \qquad (c, t\ m(t_1 var_1, \ldots, t_n var_n)\{e'\}) = findMeth(c, m) \\ n \geq 1 \qquad \sigma'' = \{var_i \mapsto v_i \mid 2 \leq i \leq n, v_i = \sigma'(var_i)\} \oplus \{var_1 \mapsto loc_1\} \oplus \{\mathtt{this} \mapsto loc'\} \\ \Pi' = \{var_i : \mathtt{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\mathtt{this} : \mathtt{var}\ c\} \\ \gamma = \mathtt{lexframe}\ \sigma''\ \Pi' \qquad loc_1 \notin dom(\mu) \qquad \mu' = \mu \oplus (loc_1 \mapsto \mathtt{eClosure}(H)\ (e, \sigma, \Pi))\end{array}}{\langle \mathbb{E}[\mathtt{invoke}(loc)], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ e'], \gamma + \rho, \mu', \psi \rangle}$$

(INVOKE-DONE)

$$\dfrac{\mathtt{eClosure}(\bullet)\ (e, \sigma, \Pi) = \mu(loc) \qquad \rho' = \sigma + \rho}{\langle \mathbb{E}[\mathtt{invoke}(loc)], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ e], \rho', \mu, \psi \rangle}$$

(UNDER)

$$\langle \mathbb{E}[\mathtt{under}\ v], \gamma + \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[v], \rho, \mu, \psi \rangle$$

(NEW)

$$\dfrac{loc \notin dom(\mu) \qquad \mu' = \mu \uplus \{loc \mapsto [c.\{f \mapsto defaultValOf(t) \mid (t\ f) \in fieldsOf(c)\}]\}}{\langle \mathbb{E}[\mathtt{new}\ c()], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu', \psi \rangle}$$

(CALL)

$$\dfrac{\begin{array}{c}loc \in dom(\mu) \qquad [c.F] = \mu(loc) \\ (c, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}) = findMeth(c, m) \qquad \sigma = \{var_i : v_i \mid 1 \leq i \leq n\} \uplus \{\mathtt{this} : loc\} \\ \Pi = \{var_i : \mathtt{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\mathtt{this} : \mathtt{var}\ c\} \qquad \gamma = \mathtt{lexframe}\ \sigma\ \Pi\end{array}}{\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ e], \gamma + \rho, \mu, \psi \rangle}$$

(CAST)

$$\dfrac{[c'.F] = \mu(loc) \qquad c' \preccurlyeq c}{\langle \mathbb{E}[\mathtt{cast}\ c\ loc], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu, \psi \rangle}$$

(GET)

$$\dfrac{loc \in dom(\mu) \qquad [c.F] = \mu(loc) \qquad v = F(f)}{\langle \mathbb{E}[loc.f], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[v], \rho, \mu, \psi \rangle}$$

(SET)

$$\dfrac{loc \in dom(\mu) \qquad [c.F] = \mu(loc) \qquad \mu' = \mu \uplus \{loc \mapsto [c.F \oplus (f \mapsto v)]\}}{\langle \mathbb{E}[loc.f = v], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[v], \rho, \mu', \psi \rangle}$$

(ASSIGN)

$$\dfrac{\sigma = envOf(\gamma) \qquad var \in dom(\sigma) \qquad \sigma' = \sigma \uplus \{var : v\} \qquad \Pi = tenvOf(\gamma) \qquad \gamma' = \mathtt{lexframe}\ \sigma'\ \Pi}{\langle \mathbb{E}[var = v], \gamma + \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[v], \gamma' + \gamma + \rho, \mu, \psi \rangle}$$

(SEQUENCE)

$$\langle \mathbb{E}[v; e], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[e], \rho, \mu, \psi \rangle$$

(DEFINE)

$$\dfrac{\begin{array}{c}\sigma = envOf(\gamma) \qquad var \notin dom(\sigma) \\ \sigma' = \sigma \uplus \{var : v_1\} \qquad \Pi = tenvOf(\gamma) \qquad \Pi' = \Pi \uplus \{var : \mathtt{var}\ t\} \qquad \gamma' = \mathtt{lexframe}\ \sigma'\ \Pi'\end{array}}{\langle \mathbb{E}[t\ var = v; e], \gamma + \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ e], \gamma' + \gamma + \rho', \mu, \psi \rangle}$$

Fig. 5. Operational Semantics of Ptolemy-I, Based on Ptolemy and MiniMAO [2, 19]

```
$hbind(loc_s, \rho, \mu, \bullet) = \bullet$
$hbind(loc_s, \rho, \mu, \vartheta + \Psi) =$
    $concat(hmatch(CT, \rho, \mu, loc_o),$
      $hbind(loc_s, \rho, \mu, \Psi))$
  where $\vartheta = [loc_o]$ is register binding

$hbind(loc_s, \rho, \mu, \theta + \Psi) =$
  if $loc_s = loc_{s'}$ then
    $concat(hmatch(CT, \rho, \mu, loc_{o'}),$
      $hbind(loc_s, \rho, \mu, \Psi))$
  else
      $hbind(loc_s, \rho, \mu, \Psi)$
  where $\theta = [loc_{o'}, loc_{s'}]$ is associate binding

$bindings(CT, c) = binds(CT, CT, c)$

$concat(\bullet, H') = H'$
$concat(h + H, H') = h + concat(H, H')$

$hmatch(CT, \rho, \mu, loc) = match(H, \rho, \mu, loc)$
  where $\mu(loc) = [c.F]$ & $H = bindings(CT, c)$

$match(\bullet, \rho, \mu, loc) = \bullet$
$match(binding + H, (\texttt{evframe}\ p'\ \rho'\ \Pi) + \rho, \mu, loc) =$
  if $p \equiv p'$ then
  let $\rho'' = \rho'$
      in let $\rho''' = \{var_i \mapsto \rho(var_i) \mid 1 \leq i \leq n\}$
      in$(\langle loc, m, \rho''' \rangle + match(H, \rho'', \mu, loc))$
    else $match(H, \rho, \mu, loc)$
  where $binding = \texttt{when}\ p\ \texttt{do}\ m$

$binds(CT, \bullet, c) = \bullet$
$binds(CT, ((t\ \texttt{evtype}\ p\{\ \ldots\ \}) + CT'), c) = binds(CT, CT', c)$
$binds(CT, ((\texttt{class}\ c\ \texttt{extends}\ c'\ \ldots\ binding_1 \ldots binding_n) + CT'), c) =$
  $concat((binding_n + \ldots + binding_1 + \bullet), binds(CT, CT, c'))$

$findMeth(c, m) = (c, t\ m(t_1 var_1, \ldots, t_n var_n)\{e\})$
  where $ct \in CT$ & $ct = \{c\ \texttt{extends}\ d,\ fld^*\ meth^*\ binding^*\}$ & $\exists i \in \{1 \ldots k\},\ meth_i = t\ m(t_1 var_1, \ldots, t_n var_n)$

$fieldsOf(c) = \{f_i \rightarrow t_i, i \in \{1 \ldots n\}\} \cup F'$
  where $ct \in CT$ & $ct = \{c\ \texttt{extends}\ d,\ fld^*\ meth^*\ binding^*\}$ & $F' = fieldsOf(d)$

$defalutValueOf(t) = \texttt{null}$   where $isType(t)$
$defalutValueOf(t) = 0$   where $!isType(t)$

$envOf(\texttt{lexframe}\ \sigma\ \Pi) = \sigma$
$envOf(\texttt{evframe}\ p\ \sigma\ \Pi) = \sigma$

$tenvOf(\texttt{lexframe}\ \sigma\ \Pi) = \Pi$
$tenvOf(\texttt{evframe}\ p\ \sigma\ \Pi) = \Pi$
```

Fig. 6. Auxiliary Functions Inspired by Work of Rajan and Leavens on Ptolemy [19]

Figure 6, then all of its formal parameter bindings are extracted from σ' and added to σ''. After the current event handler is executed, it is removed from the list of event handler records and a new event closure is created and stored in a new location loc_1 in the store. As the first parameter in every event handler method call is an event closure location, then the binding of the first formal parameter and the event closure is put on top of the stack. This process is run repeatedly, until there are no more records in the event handlers records list. At this time the body of the event is executed as it is shown in Figure 5 in (INVOKEDONE).

To compute the list of handler records, we need to know which classes have event bindings. Event binding expression **`when`** p **`do`** m in a class c states that whenever an event of type p is announced, the handler method m should execute if this class or its individual instances are registered with the subject announcing the event. To figure out elements of the handler records list, the $hbind$ function shown in Figure 6 goes through the list of binding records ψ recursively and checks the individual records in the list. When encountering a register binding ϑ, the control is handed over to the $hmatch$ function. If an associate binding record θ is encountered and the subject instance publishing the event and subject instance in the associate binding record are the same, again $hmatch$ does the rest of the processing and concatenates its results with current list of handler records. If none of these cases happen, then the record on the list will be ignored. Function $hmatch$ searches CT for event bindings of an specific class c. It also takes care of inheritance when looking for class bindings. If the declaration in CT is an event type declaration of the form t **evtype** $p\{\ \}$ it is ignored, as it has no bindings. But if it is a class declaration, its event bindings plus its super class bindings are added to the result.

The $hmatch$ function determines, for a particular object loc, what bindings declared in the class of the object referred to by loc are applicable. It looks up the location loc in the store, extracts the class that the object loc refers to, and uses that class to obtain a list of potential bindings. This list is filtered using $match$, which matches p against a particular event on the stack. Each matched binding generates a handler record, recording the active object (which will act as a receiver when the handler method is called), the handler method's name and an environment. The environment is obtained from the **evframe**. This environment is also restricted to contain just those mappings that are for names in the declared formal arguments of the binding.

Standard OO expressions and their semantics are similar to those discussed in more detail in [18] with the exception that, in Ptolemy-I, we use an ordered list ψ of associate/register binding records to keep track of subject-observer instances on individual basis. As it can be seen in Figure 5 none of the standard OO expressions change or specifically use ψ. Auxiliary functions used in semantics of Ptolemy-I's expressions are defined in Figure 6

5 Type System of Ptolemy-I

Ptolemy-I's type checking attributes are shown in Figure 4 and rules are shown in Figure 7.

Class table CT along with some auxiliary functions are used in the type checking rules. We assume that names declared at top levels of the program are distinct and there is no cyclic inheritance relationship. Relation $c' \preccurlyeq c$ means c' is the subtype of c. As in the work on Ptolemy [19], it is the reflexive transitive closure of the declared subclasses relationships. In the defined typing attributes, *OK* and *OK in* c are used to denote that a term is well typed or well typed in the context of class c, respectively. Π represents the typing environment. There are two kinds of declarations in the program that should be type checked, classes and events.

A class type checks if all of its fields, methods and binding definitions are well typed and the class extends a valid class type . When type checking a class, we don't allow overriding of super class fields. The type checking rules for fields and methods are standard. A binding declaration **when** p **do** m type checks if the handler method is a valid method, the return type of the handler method m is the same as the return type of the event type declaration p, and the arguments of the handler method match both in names and types with the context variables of p.

An **evtype** type checks if its return type and all of its context variables have a defined type. An event expression **event** `p { e }` type checks if (a) the body of the event type checks, (b) the event context variables are well defined in the typing environment, (c) the event context variables have the same type as their declared type in the event type declaration p, and (d) the type of the body expression is a subtype of the declared return type for the event.

An associate expression $\texttt{\textbf{associate}}(e_0, e_1)$ type checks if e_0 and e_1 are well typed. A register expression $\texttt{\textbf{register}}(e)$ type checks if e is well typed and its type is the same as that of e. An invoke expression $\texttt{\textbf{invoke}}(e)$ type checks if e is of type $\texttt{\textbf{thunk}}\ c$ ensuring that e is an event closure. The under expression $\texttt{\textbf{under}}\ e$ type checks if the type of the expression e is valid. The type of $\texttt{\textbf{under}}\ e$ is the same as type of e.

Some auxiliary functions are used in the type checking rules and can be seen in Figure 7. *isClass* ensures that the given parameter is a valid class name which exists in the CT. *isThunkType* checks that a given class name is valid and also is an event type. Finally, *isType* returns true if its argument is a valid class or a valid event type.

6 Type Soundness

The proof of soundness of Ptolemy-I's type system follows the standard preservation and progress argument [32]. Progress says that the evaluation of a term does not get stuck, as the term could either be a value or there is a evaluation rule for it. Preservation emphasizes on preserving expression type during reduction phases and consistency of typing environment Π and store μ.

Progress:

Theorem 1. *(Progress)*
For a well-typed expression e, stack ρ, store μ, list ψ and type environment Π which is consistent with μ. If $\Pi \vdash e : t$ then either

- $e = loc$ *and* $loc \in dom(\mu)$ *or* $e = \texttt{\textbf{null}}$
- $\langle e, \rho, \mu \rangle \hookrightarrow \langle e', \rho', \mu' \rangle$

Proof: (a) If expression e is a location value (loc) and $\Pi \vdash loc : t$, then according to (T-LOC), $loc \in dom(\Pi)$, and assuming the consistency of store and typing environment, $\Pi \approx (\mu, \rho)$, then $loc \in dom(\mu)$ as $dom(\Pi) = dom(\mu)$.

(b) In the case where the expression e is not a value, we consider the evaluation rules case by case for providing the proof. We proceed with the induction of derivation of expression e. Assuming that all the sub-terms of expression e have progress. Induction hypothesis (IH) assumes all sub-terms of e to be well-typed and we use this assumption in our proof frequently. We define configuration cg as $c = \langle \mathbb{E}[e], \rho, \mu, \psi \rangle$

- *case 1.* $e = \mathbb{E}[\texttt{\textbf{associate}}(loc, loc')]$. Based on IH, sub-terms loc and loc' are well-typed, as it is also ensured by (T-ASSOCIATE) rule; Therefore the configuration cg evolves according to (ASSOCIATE).
- *case 2.* $e = \mathbb{E}[\texttt{\textbf{register}}(loc)]$. Based on the IH, loc is well-typed as it is also ensured by (T-REGISTER); Therefore the configuration cg evolves according to (REGISTER).
- *case 3.* $e = \mathbb{E}[\texttt{\textbf{event}}\ p\ \{e\}]$. Sub-term e is well-typed based on IH. Rule (T-EVENT) ensures $\Pi \vdash e : \texttt{\textbf{exp}}\ c$; Therefore configuration cg evolves according to (EVENT).

- *case 4.* $e = \mathbb{E}[\texttt{invoke}(\mathit{loc})]$. sub-term loc is well-typed based on IH besides that (T-INVOKE) ensures $\Pi \vdash \mathit{loc} : \texttt{exp}\ (\texttt{thunk}\ c)$; Therefore configuration cg evolves based on (INVOKE) reduction rule.
- *case 5.* $e = \mathbb{E}[\mathit{loc}.m(v_1, \ldots, v_n)]$. Based on the IH, sub-term loc is well-typed $\Pi \vdash \mathit{loc} : c$. According to (T-LOC), $\mathit{loc} \in \mathit{dom}(\Pi)$ and as $\Pi \approx (\mu, \rho)$ then $\mathit{loc} \in \mathit{dom}(\mu)$.
  Again based on IH, method m is well-typed therefore, type of its body e, is the same as its return type. Meaning that cg evolves by (CALL).
- *case 6.* $e = \mathbb{E}[\mathit{loc}.f]$. Based on IH loc and $\mathit{loc}.f$ both are well-typed sub-terms, $\mu(\mathit{loc}) = [c.F]$. Having $\mathit{loc}.f$ as a well-typed expression implies that $f \in \mathit{fieldsOf}(c)$ based on (T-GET) type checking rule; Therefore, cg evolves by (GET).
- *case 7.* $e = \mathbb{E}[\mathit{loc}.f = v]$. The argument is the same as the argument made for case 6.
- *case 8.* $e = \mathbb{E}[\texttt{cast}\ t\ \mathit{loc}]$. Based on IH, sub-term loc is well-typed implying such that $\Pi \vdash \mathit{loc} : c'$ and $\mu(\mathit{loc}) = [c'.F]$. On the other hand (CAST) ensures $c' \preccurlyeq c$ then cg evolves correctly by (CAST).
- *case 9.* $e = \mathbb{E}[\texttt{new}\ c()]$. Based on IH all sub-terms are well-typed, rendering the case trivial, since as long as we have $\mathit{isClass}(c)$ to be true, configuration cg evolves according to (NEW). Consistency of store and typing environment, assures that newly created loc is of type c.
- *case 10.* $e = \mathbb{E}[var\ = v]$. Based on the IH, sub-terms var and v both are well-typed. On the other hand, (T-ASSIGN) ensures their types are the same. Therefore, configuration cg evolves according to (ASSIGN).
- *case 11.* $e = \mathbb{E}[\texttt{under}\ v]$. Based on the IH, sub-term v is well-typed as it is also assured by (T-UNDER). Therefore, configuration cg evolves by (UNDER).
- *case 12.* $e = \mathbb{E}[t\ var = v; e]$. Based on the IH, sub-terms var, v and e are all well-typed. On the other hand, (T-DEFINE) ensures the type of var and v are the same. Therefore, configuration cg evolves according to (DEFINE).
- *case 13.* $e = \mathbb{E}[v_1; v_2]$. Based on IH, both terms v_1 and v_2 are well-typed. Therefore, configuration cg is evolved based on the rule (SEQUENCE).

Preservation:

Theorem 2. *(Subject Reduction)*
Given an expression e, stack ρ, store μ and a typing environment Π, consistent with the store and stack $\Pi \approx (\mu, \rho)$, if $\Pi \vdash e : t$ and $\langle e, \rho, \mu \rangle \hookrightarrow \langle e', \rho', \mu' \rangle$, then there exist Π' and t' such that $\Pi' \approx (\mu', \rho')$, $\Pi \vdash e' : t'$ and $t' \preccurlyeq t$.

As it can be seen to prove preservation, it should be proved that (a) reduction preserves the type and (b) typing environment is consistent with store and stack, $\Pi \approx (\rho, \mu)$.

Consistency of type environment Π and store μ is defined as follows [2]:

- 1. $\mu(\mathit{loc}) = [c.F] \Rightarrow$
  (a) $\Pi(\mathit{loc}) = c$

(b) $\mathit{dom}(F) = \mathit{dom}(\mathit{fieldsOf}(c))$
(c) $\mathit{rng}(F) = \mathit{dom}(\mu) \cup \{\mathbf{null}\}$
(d) $\forall f \in \mathit{dom}(F).\{F(f) = \mathit{loc}' \;\&\; \mathit{fieldsOf}(c)(f) = u \;\&\; \mu(\mathit{loc}') = [c'.F']\} \Rightarrow t' \preccurlyeq u$

– 2. $\mathit{loc} \in \mathit{dom}(\mathit{\Pi}) \Rightarrow \mathit{loc} \in \mathit{dom}(\mu)$
– 3. $\mathit{dom}(\mu) \subseteq \mathit{dom}(\mathit{\Pi})$

Before proceeding with the proof, we need a lemma, replacement lemma, which is taken from Clifton's work on MiniMAO [2]

Lemma 1. *(Replacement)* $\mathit{\Pi} \vdash \mathbb{E}[e] : t,\ \mathit{\Pi} \vdash e : t' \;\&\; \mathit{\Pi} \vdash e' : t' \Rightarrow \mathit{\Pi} \vdash \mathbb{E}[e'] : t$

To show preservation property in Ptlemy-I, we show it case by case for all of the evaluation rules defined for the language expressions.

(ASSOCIATE)

$$\frac{\mathit{loc} \in \mathit{dom}(\mu) \qquad \mathit{loc}' \in \mathit{dom}(\mu) \qquad \theta = [\mathit{loc}, \mathit{loc}'] \qquad \psi' = \theta + \psi}{\langle \mathbb{E}[\mathbf{associate}(\mathit{loc}, \mathit{loc}')], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[\mathit{loc}], \rho, \mu, \psi'\rangle}$$

– Proof of consistency relation $\mathit{\Pi} \approx (\rho, \mu)$ is trivial, as neither the stack nor the store changes in this rule.
– Now we show that $\mathit{\Pi} \vdash \mathbb{E}[\mathbf{associate}(\mathit{loc}, \mathit{loc}')] : t \Rightarrow \mathit{\Pi}' \vdash \mathbb{E}[\mathit{loc}] : t$. Based on the (T-ASSOCIATE), $\mathit{\Pi}' \vdash \mathbf{associate}(\mathit{loc}, \mathit{loc}') : \mathbf{exp}\, c_0$ where $\mathit{\Pi}' \vdash \mathit{loc} : \mathbf{exp}\, c_0$. Using (REPLACEMENT) lemma, we replace $\mathbf{associate}(\mathit{loc}, \mathit{loc}')$ with loc in $\mathit{\Pi} \vdash \mathbb{E}[\mathbf{associate}(\mathit{loc}, \mathit{loc})] : t$, as they have the same type, concluding that $\mathbb{E}[\mathit{loc}] : t$.

(REGISTER)

$$\frac{\mathit{loc} \in \mathit{dom}(\mu) \qquad \vartheta = [\mathit{loc}] \qquad \psi' = \vartheta + \psi}{\langle \mathbb{E}[\mathbf{register}(\mathit{loc})], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[\mathit{loc}], \rho, \mu, \psi'\rangle}$$

– Proof of consistency $\mathit{\Pi} \approx (\rho, \mu)$ is the same as (ASSOCIATE) as neither stack nor store changes.
– Now we show $\mathit{\Pi} \vdash \mathbb{E}[\mathbf{register}(\mathit{loc})] : t \Rightarrow \mathit{\Pi}' \vdash \mathbb{E}[\mathit{loc})] : t$. Based on the (T-REGISTER), $\mathit{\Pi}' \vdash \mathbf{register}(\mathit{loc}) : \mathbf{exp}\, c$ where $\mathit{\Pi}' \vdash \mathit{loc} : \mathbf{exp}\, c$. Using (REPLACEMENT), $\mathbf{register}(\mathit{loc})$ is replaced with loc in $\mathit{\Pi} \vdash \mathbb{E}[\mathbf{register}(\mathit{loc})] : t$ as they both have the same type, concluding that $\mathbb{E}[\mathit{loc})] : t$.

(EVENT)

$$\frac{\begin{array}{c}(c\ \mathbf{evtype}\ p\{t_1\ var_1, \ldots, t_n\ var_n\}) \in CT \\ \sigma = envOf(\gamma) \qquad \sigma' = \{var_i \mapsto v_i \mid v_i = \sigma(var_i)\} \qquad \mathit{loc} \notin \mathit{dom}(\mu) \qquad \mathit{loc}' = \sigma(\mathbf{this}) \\ H = hbind(\mathit{loc}', \gamma' + \gamma + \rho, \mu, \psi) \qquad \mathit{\Pi}' = \{var_i : \mathbf{var}\ t_i \mid 1 \le i \le n\} \uplus \{\mathit{loc} : \mathbf{var}\ (\mathbf{thunk}\ c)\} \\ \gamma' = \mathbf{evframe}\ p\ \sigma'\ \mathit{\Pi}' \qquad \mu' = \mu \oplus (\mathit{loc} \mapsto \mathbf{eClosure}(H)\ (e, \sigma, \mathit{\Pi}))\end{array}}{\langle \mathbb{E}[\mathbf{event}\ p\ \{e\}], \gamma + \rho, \mu, \psi\rangle \qquad \hookrightarrow \langle \mathbb{E}[\mathbf{under}\ (\mathbf{invoke}(\mathit{loc}))], \gamma' + \gamma + \rho, \mu', \psi\rangle}$$

– To show the consistency, let $\mathit{\Pi}' = \mathit{\Pi} \uplus \mathit{loc} : \mathbf{thunk}\, c$ as it is ensured by (T-INVOKE). Having $\mathit{loc} \in \mathit{dom}(\mathit{\Pi}')$, $\mathit{loc} \in \mathit{dom}(\mu')$ and the typing environment consistency with stack and store, $\mathit{\Pi} \approx (\mu, \rho)$, consistency $\mathit{\Pi} \approx (\mu', \rho')$ holds.

- Now we show $\Pi \vdash \mathbb{E}[\textbf{event}\ p\{e\}] : t \Rightarrow \Pi' \vdash \mathbb{E}[\textbf{under}\ (\textbf{invoke}(loc))] : t$. Based on (T-EVENT) we know $\Pi' \vdash \textbf{event}\ p\{e\} : c$ and $\Pi' \vdash loc : \textbf{thunk}\ c$. On the other hand, based on (T-INVOKE) $\Pi' \vdash \textbf{invoke}(loc) : c$. Using (REPLACEMENT), we replace $\textbf{event}p\{e\}$ with $\textbf{invoke}(loc)$ in $\mathbb{E}[\textbf{event}p\{e\}] : t$ and conclude that $\mathbb{E}[\textbf{invoke}(loc)] : t$. Type checking rule (T-UNDER) ensures $\mathbb{E}[\textbf{under}\ \textbf{invoke}(loc)] : t$.

(INVOKE)

$$\frac{\begin{array}{c}\textbf{eClosure}((\langle loc', m, \sigma'\rangle + H))\ (e, \sigma, \Pi) = \mu(loc) \\ [c.F] = \mu(loc') \qquad (c, t\ m(t_1 var_1, \ldots, t_n var_n)\{e'\}) = findMeth(c, m) \\ n \geq 1 \qquad \sigma'' = \{var_i \mapsto v_i \mid 2 \leq i \leq n, v_i = \sigma'(var_i)\} \oplus \{var_1 \mapsto loc_1\} \oplus \{\textbf{this} \mapsto loc'\} \\ \Pi' = \{var_i : \textbf{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\textbf{this} : \textbf{var}\ c)\} \\ \gamma = \textbf{lexframe}\ \sigma''\ \Pi' \qquad loc_1 \notin dom(\mu) \qquad \mu' = \mu \oplus (loc_1 \mapsto \textbf{eClosure}(H)\ (e, \sigma, \Pi))\end{array}}{\langle \mathbb{E}[\textbf{invoke}(loc)], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[\textbf{under}\ e'], \gamma + \rho, \mu', \psi\rangle}$$

- Showing consistency relation is the similar to (EVENT).
- Now we should show $\Pi \vdash \mathbb{E}[\textbf{invoke}(e)] : t \Rightarrow \Pi' \vdash \mathbb{E}[\textbf{under}\ \ e'] : t$. Based on (T-INVOKE) we know $\Pi' \vdash \textbf{invoke}(e) : c$ where $\Pi' \vdash loc : \textbf{thunk}\ c$. On the other hand, based on (T-INVOKE) $\Pi' \vdash \textbf{invoke}(loc) : c$. On the other hand, whatever handler method returns which has the same type of its body e', is of type c. Therefore using (REPLACEMENT) we replace $\textbf{invoke}(e)$ with $\textbf{under}\ \ e'$ in $\mathbb{E}[\textbf{invoke}(e)] : t$ and conclude that $\mathbb{E}[\textbf{under}\ \ e'] : t$. Type checking rule (T-UNDER) ensures $\mathbb{E}[\textbf{under}\ \ e')] : c$ as well.

(CALL)

$$\frac{\begin{array}{c} loc \in dom(\mu) \qquad [c\,.\,F] = \mu(loc) \\ (c, m(t_1\ var_1, \ldots, t_n\ var_n)\{e\}) = findMeth(c, m) \qquad \sigma = \{var_i : v_i \mid 1 \leq i \leq n\} \uplus \{\textbf{this} : loc\} \\ \Pi' = \{var_i : \textbf{var}\ t_i \mid 1 \leq i \leq n\} \uplus \{\textbf{this} : \textbf{var}\ c\} \qquad \gamma = \textbf{lexframe}\ \sigma\ \Pi' \end{array}}{\langle \mathbb{E}[loc.m(v_1, \ldots, v_n)], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[\textbf{under}\ e], \gamma + \rho, \mu, \psi\rangle}$$

- First we show that $\Pi' \approx (\mu', \rho')$. Let $\Pi' = \Pi$. As we know store does not change ($\mu' = \mu$). Therefore, using assumption $\Pi \approx (\mu, \rho)$ we have $\Pi \approx (\mu', \rho')$.
  As the program type checks based on (T-PROGRAM), consequently its classes and methods in classes type check as well. Based on (T-METHOD) we choose $\Pi' = \{var_1 : t_1, \ldots, var_i : t_i, \ldots, var_n : t_n, \textbf{this} : c\}$. Method type checking rule (T-METHOD) also specifies a type t' for the body e of the method, $\Pi' \vdash e : \textbf{exp}\ t'$ and $t' \preccurlyeq t$.
- We show $\Pi \vdash \mathbb{E}[loc.m(v_1, \ldots, v_n)] : t \Rightarrow \Pi' \vdash \mathbb{E}[\textbf{under}\ \ e] : t$. Rule (T-CALL) ensures that, the returning the method is called on the object with the right type, and the type of method body e is sub-type of returning type of the method. This paves the way for using (REPLACEMENT) and replace $loc.m(v_1, \ldots, v_n)$ with $\textbf{under}\ e$ in $\mathbb{E}[loc.m(v_1, \ldots, v_n)]$, as they both have the same type, concluding that $\mathbb{E}[\textbf{under}\ \ e] : t$.

(GET)

$$\frac{loc \in dom(\mu) \qquad [c.F] = \mu(loc) \qquad v = F(f)}{\langle \mathbb{E}[loc.f], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[v], \rho, \mu, \psi\rangle}$$

- Showing $\Pi' \approx (\mu', \rho')$ is trivial. As it can be seen, neither stack, nor store, nor typing environment changes.

– Now we show that $\varPi \vdash \mathbb{E}[loc.f] : t \Rightarrow \varPi' \vdash \mathbb{E}[v] : t$. Using (T-LOC) and the assumption $\varPi \approx (\mu, \rho)$, we have $\varPi(loc) = c$. On the other hand, (T-GET) ensures that type of field $\mathit{fieldsOf}(c)(f) = v$ is the same as c; Therefore, we can easily replace $\mathbb{E}[loc.f]$ with $\mathbb{E}[v]$ using (REPLACEMENT), concluding that $\mathbb{E}[v] : t$. The case for (SET) is similar.

(CAST)
$$\frac{[c'.F] = \mu(loc) \qquad c' \preccurlyeq c}{\langle \mathbb{E}[\mathtt{cast}\ c\ loc], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu, \psi\rangle}$$

– Proving consistency is trivial as neither stack nor store changes.
– Now we show that $\varPi \vdash \mathbb{E}[\mathtt{cast}\ t\ e] : t \Rightarrow \varPi' \vdash \mathbb{E}[loc] : t$. Rule (CAST) assures that type of expression e is the subtype of class t, while (T-CAST) makes sure that t is a valid type. Therefore we replace $\mathtt{cast}\ t\ e$ with loc in $\mathbb{E}[\mathtt{cast}\ t\ e] : t$, having $\mathbb{E}[loc] : t$.

(NEW)
$$\frac{loc \notin dom(\mu) \qquad \mu' = \mu \uplus \{loc \mapsto [c.\{f \mapsto \mathit{defaultValOf}(t) \mid (t\ f) \in \mathit{fieldsOf}(c)\}]\}}{\langle \mathbb{E}[\mathtt{new}\ c()], \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[loc], \rho, \mu', \psi\rangle}$$

– Let $\varPi' = \varPi \uplus loc : c$. We have $loc \in dom(\varPi')$, $loc \in dom(\mu')$ and the assumption that $\varPi \approx (\mu, \rho)$; therefore parts 2 and 3 of consistency definition hold, because $loc \notin dom(\mu), (\varPi \approx \mu) \Rightarrow loc \notin dom(\varPi)$. As $\mu'(loc) = [c.F]$, $\varPi'(loc) = c$, $dom(F) = dom(\mathit{fieldsOf}(c))$, $rng(F) = \{\mathtt{null}\} \subseteq dom(\mu) \cup \{\mathtt{null}\}$, so part 1 holds too. Therefore we have $\varPi' \approx (\mu', \rho')$, where $\rho' = \rho$, as the stack doesn't change here.
– Now we show that $\varPi \vdash \mathbb{E}[\mathtt{new}\ c()] : t \Rightarrow \varPi' \vdash \mathbb{E}[loc] : t$. We know $\varPi' \vdash \mathtt{new}\ c() : c$ and $\varPi' \vdash loc : c$, so using replacement lemma, we replace $\mathtt{new}\ c()$ with loc in $\mathbb{E}[\mathtt{new}\ c()] : t$ and conclude that $\mathbb{E}[loc] : t$.

(DEFINE)
$$\frac{\begin{array}{c}\sigma = \mathit{envOf}(\gamma) \qquad var \notin dom(\sigma)\\ \sigma' = \sigma \uplus \{var : v_1\} \qquad \varPi = \mathit{tenvOf}(\gamma) \qquad \varPi' = \varPi \uplus \{var : \mathtt{var}\ t\} \qquad \gamma' = \mathtt{lexframe}\ \sigma'\ \varPi'\end{array}}{\langle \mathbb{E}[t\ var = v; e], \gamma + \rho, \mu, \psi\rangle \hookrightarrow \langle \mathbb{E}[\mathtt{under}\ e], \gamma' + \gamma + \rho', \mu, \psi\rangle}$$

– Proving consistency of stack and store is similar to the approach we have takne in the proof for rule (CALL). Store doesn't change and we have taken care of new typings added to typing environment.
– Now we show that $\varPi \vdash \mathbb{E}[t\ var = e_1; e_2] : t \Rightarrow \varPi' \vdash \mathbb{E}[\mathtt{under}\ \ e_2] : t$. Rule (T-DEFINE) assures that type of expression e_1 is the subtype of class t and the return type of the expression is t which is the same as type of expression e_2. And having the IH, we know, all of the sub-terms are well-typed. On the other hand, (T-CAST) makes sure the return type of the expression $\mathtt{under}\ \ e_2$ is the same as type e_2. Therefore, we can easily replace $t\ var = e_1; e_2$ with $\mathtt{under}\ \ e_2$ in $\mathbb{E}[t\ var = e_1; e_2] : t$, having $\mathbb{E}[\mathtt{under}\ \ e_2] : t$. The similar argument applies to (ASSIGN).

(SEQUENCE)
$\langle \mathbb{E}[v_1; v_2], \rho, \mu, \psi \rangle \hookrightarrow \langle \mathbb{E}[v_2], \rho, \mu, \psi \rangle$

- Proving consistency of stack and store is trivial because stack and store, both remain the same and intact.
- Now we show that $\Pi \vdash \mathbb{E}[e_1; e_2] : t \Rightarrow \Pi' \vdash \mathbb{E}[e_2] : t$. Again based on the IH, all of sub-terms e_1 and e_2 are well-typed. Rule (T-SEQUENCE) assures that type of expression e_2 is the same as the return type of the whole sequence expression (t). Therefore, based on (SEQUENCE) we replace $e_1; e_2$ with e_2 in $\mathbb{E}[t\ var = e_1; e_2] : t$ as both of them have the same type, concluding $\mathbb{E}[e_2] : t$.

7 Initial Assessment of the Language Design

To provide an initial assessment of our language design with instance-level quantified, typed events, we apply it to an example Graph System (GS) used by Rajan and Sullivan [22]. First, we present an overview of the example and integration requirements. We then analyze Ptolemy-I's implementation of this system.

Overview. GS exploits the Model-View-Controller (MVC) architecture with several classes in each layer. Figure 8 shows the contributing classes of GS in boxes with plain white background. In the view layer, classes `PS` (PointSet) and `LS` (LineSet) are responsible for keeping track of a set of `Points` and `Lines`, respectively, depicted on the user interface. The `UI` class is responsible for the user interface. In the model layer, classes `VS` (VertexSet) and `ES` (EdgeSet) model each `Vertex` and all `Edges` associated with points and lines in the view layer.

Example Graph System Requirement. A very natural requirement of GS system is *consistency*. The system should remain consistent when lines are added or removed from the UI. In other words, if a line is removed in the view layer, its associated edge in the model layer should also be removed. Similarly, lines added to the view layer should be added to the model layer.

The controller classes `VS-PS` (VertexSet-PointSet) and `ES-LS` (EdgeSet-LineSet) synchronize the model and the view, keeping them consistent. For example `ES-LS` is responsible for keeping `ES` consistent with `LS`. Whenever a `Line` object is removed from `LS`, the associated `ES` removes the associated edge. Class `Lazy` synchronizes `UI` in the view layer and `G` (Graph) in the model layer in two different modes: *agile* and *lazy*. In agile mode, changes are synchronized as soon as they occur in the UI, but in lazy mode changes are buffered and synchronized in batches.

Ptolemy-I's Solution and Analysis. One solution to implement the consistency requirement in an integrated system is to use subject-observer behavioral patterns [7]. In event based systems, objects like `LS` announce events and others like `LS-ES`, interested in those events, receive notification to handle them. Ptolemy [19], with support for quantified, typed events is a good candidate to implement such event-based integrated systems. But the implementation of this use case in Ptolemy (that lacks the support for instance-level typed events) would not be efficient in the case where there are numerous `LS` and `ES` objects in the system. In Ptolemy, whenever a `Line` object is removed from a line set `LS`, an event, say `LineRemove`, is fired to inform interested `ES` objects to update themselves. Ptolemy's implementation would suffer overhead due to every instance of `ES` checking if the line removed was in its model.

Figure 8 shows the architectural design of the GS system using quantified, typed events. In this figure, when an event `LineRemove` is announced by an `LS` object, all registered `ES-LS` instances are notified and handle the event by calling the handler method `LineRemoveEventHandler`. Likewise, whenever an edge is removed from an `ES` instance an event of type `EdgeRemoved` is announced and handled by `ES-LS` by calling `EdgeRemoveEventHandler`. To model the lazy and agile modes, two events `Lazy-VC` and `Lazy-CM` are created along with their handlers. Whenever the synchronization mode changes from eager to lazy or vice-versa, first the event `Lazy-VC` is fired and consequently the event `Lazy-CM` is fired, as they are designed to communicate between `UI`, `Lazy` and `G` objects.

There are two approaches to relate `LS` objects with `ES` objects in Ptolemy. Either an `ES` object is related to all instances of `LS` through plain **`register`** expression or the relation between individual instances of `ES` and the associated `LS` objects is kept track of through a combination of using **`register`** and a map-like structure. Both of these approaches have their problems. The first one could be inefficient when the system has a lot of `LS` objects. Announcement of the `LineRemove` event by any of them would result in the notification of all `ES` instances that are registered to receive notification for `LineRemove`. The second solution, although not suffering from this problem, adds the overhead of keeping the map of relations between `LS` and `ES` objects updated. As it can be seen, both of these solutions are inefficient in the sense that they both have unnecessary processing in terms of unneeded notifications or updating the subject-observer instances relationship map.

Our solution in Ptolemy-I, which solves the above-mentioned problems and reduces the inefficiency imposed on the system by type-level events in Ptolemy, is to allow individual instances of the `LS` and `ES` types to be related through support for instance-level typed events using **`associate`**. With instance-level typed events, when an instance of `LS` announces an event of type `LineRemove` only `ES` instances that have been related to that specific `LS` instance will be notified, reducing the overhead of unnecessary event notifications that was happening in Ptolemy. As the language itself takes care of keeping track of related subject-observer instances, there is no need to maintain a separate map in the implementation. The support for instance-level quantified, typed events in Ptolemy-I thus further improves the separation of integration concerns in GS.

8 Related Work

Our work has two facets, one concerning quantified, typed events and the other concerning instance-level features for modularization techniques. Compared to instance-level typed events which is a new idea to our knowledge with no prior work, class-level quantified typed events is not. Rajan and Leavens have introduced a language, Ptolemy [19], which supports quantified typed, events to improve separation of concerns and more efficient implementation of integrated systems. Ptolemy also is a solution for some of the problems incurred in Aspect Oriented (AO) and Implicit Invocation (II) languages. The inability of II languages to address a broad range of events and limitations of AO languages such as their dependence on syntactical structure of the language and delimited set of available implicit events are problems solved in Ptolemy. Although instance-level typed events have no prior work, there are several works that have introduced instance-level aspects in AO languages, but have not addressed all problems.

Rajan and Sullivan proposed runtime instance-level aspect weaving [21]. They also extended C# with AspectJ-like constructs, first class aspect instances and instance-level advising [20], paving the way to implement the Mediator behavioral pattern [28] using aspect instances and instance-level advising while improving modularity. Pierce and Noble have introduced Relationship Aspects [15]. They implemented object relationships in a library of aspects in AspectJ, moving the the code handling the object's relations out of the objects and leaving them more modularized, maintainable and reusable without sacrificing efficiency. A shortcoming of their approach is that although they factor out the object relationship code into relationship aspects, since they have no mechanism supporting instance-level aspects there is no way to relate object instances using relationship aspects and they only work at the type(class)-level. Sakurai *et al.* proposed a way to associate an aspect instance with a group of objects to solve the shortcoming of the per-object aspect model of languages like AspectJ for directly supporting behavioral relationships in integrated systems [26]. Rajan and Sullivan proposed a programming language with classes and aspects unified, classpects [23], and use classpects as basic units of modularity, providing the ability to create instances of aspects and have instance-level advising. Rajan later showed that the implementation of some of Gang-of-Four (GOF) patterns using classpect constructs are improved and without degrading the remaining patterns [17]. Garcia *et al.* reported the same result regarding modularity and separation of concerns improvement [8] as Hannemann and Kiczales [9] do, improving modularity in 17 out of 23 of GOF patterns.

Other works have tried to solve the need for instance-level aspects by introducing changes in the advising model of existing AO languages. Taner proposed deployment strategies to change the advising model of AO languages and make it more customizable [5]. Deployment strategies precisely specify the scope of a deployed aspect at runtime. Hoffman and Eugster investigate a model with base code being aware of cross cutting aspects [10] and extend AspectJ with constructs that allow base code and aspects to cooperate in new ways like advising arbitrary blocks of code, explicitly parameterizing aspects, base code specifying where to apply an aspect, and enforcing new constraints by advice on base code. Rho *et al.* introduced fine grained point cuts to empower the AO languages advising model by exposing a larger set of join points [25]. But aspects are not the only way to manage behavioral relationships between objects. Bierman and Wren proposed a language construct for defining the relationship between classes and to manipulate relationship instances [1]. The problem here again, is that the relationship definition is at the type-level and not at the instance-level.

9 Future Work and Conclusions

Integrated systems must modularize behavioral relationships between several logically unrelated objects, such as debuggers, editors, compilers, etc. These behavioral relationships control the actions, control, and states of various components to implement the overall behavior of the system. Previous works in the field of Aspect-oriented (AO) languages have investigated instance-level support for separating these integration concerns. The language Ptolemy [19] aims to solve problems of such AO languages, such as fragile pointcuts [19, 27, 30], by adding quantified, typed events. The events added in Ptolemy however operate on a type(class)-level and are not directly sufficient to support such integration concerns. Instance-level support must be manually added to the system, adding extra complexity for the programmer. There is also added overhead incurred due to the event handlers executing more often than needed.

Our solution instead uses Ptolemy as the base language, which solves the problems associated with using an AO base language. We extend Ptolemy's language to add new syntax in the form of *associate* expression, which allows for the association of event publishers (subjects) to subscribers (observers) at the instance level. We also, to the best of our knowledge, give the first formal semantics of such instance-level features in a language and prove the type soundness of our approach. We also have implemented a proof of concept interpreter for our new language Ptolemy-I as well as shown an example for handling `Model` relationships and a `Graph System` example. The `Graph System` example shows that our language is able to implement complicated use cases that have been used to justify the need for instance-level support [20]. Our initial assessment of the implementation shows that instance-level quantified, typed events improve the separation of integration concerns over previous language design proposals.

Many directions for investigation remain to be investigated. Primary among them are an implementation and a detailed performance evaluation of Ptolemy-I's design as an extension of Java. Such implementation would then pave the way for a large-scale empirical evaluation by us and others, e.g. using GOF design patterns [7].

Note : Ptolemy-I's interpreter and examples are available at `http:\\www.cs.iastate.edu\~ptolemy\instance`.

References

1. G. Bierman and A. Wren. First-class relationships in an object-oriented language. In *ECOOP*, pages 262–282, 2005.
2. C. Clifton. A design discipline and language features for modular reasoning in aspect-oriented programs. Technical Report 05-15, Iowa State University, Jul 2005.
3. C. Clifton and G. T. Leavens. MiniMAO1: an imperative core language for studying aspect-oriented reasonings. *Sci. Comput. Program.*, 63(3):321–374, 2006.
4. C. Clifton and G. T. Leavens. MiniMAO$_1$: Investigating the semantics of proceed. *Sci. Comput. Programming*, 63(3):321–374, 2006.
5. Éric Tanter. Expressive scoping of dynamically-deployed aspects. In *AOSD*, pages 168–179, 2008.
6. M. Flatt, S. Krishnamurthi, and M. Felleisen. A programmer's reduction semantics for classes and mixins. In *Formal Syntax and Semantics of Java*, pages 241–269, 1999.

7. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., 1995.
8. A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. Modularizing design patterns with aspects: a quantitative study. In *AOSD '05*, pages 3–14, 2005.
9. J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02*, pages 161–173, 2002.
10. K. Hoffman and P. Eugster. Bridging Java and AspectJ through explicit join points. In *PPPJ '07*, pages 63–72, 2007.
11. A. Igarashi, B. Pierce, and P. Wadler. Featherweight Java: A minimal core calculus for Java and GJ. In *OOPSLA '99*, pages 132–146, 1999.
12. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP*, pages 327–353, Jun 2001.
13. D. C. Luckham, J. J. Kennedy, L. M. Augustin, J. Vera, D. Bryan, and W. Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–54, Apr 1995.
14. M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD*, 2003.
15. D. J. Pearce and J. Noble. Relationship aspects. In *AOSD*, pages 75–86, 2006.
16. H. Rajan. *Unifying Aspect- and Object-Oriented Program Design*. PhD thesis, The University of Virginia, Charlottesville, Virginia, August 2005.
17. H. Rajan. Design pattern implementations in eos. In *PLoP '07, Conference on Pattern Languages of Programs*, September 2007.
18. H. Rajan and G. T. Leavens. Quantified, typed events for improved separation of concerns. Technical Report 07-14, Iowa State University, Department of Computer Science, July 2007.
19. H. Rajan and G. T. Leavens. Ptolemy: A language with quantified, typed events. In *ECOOP '08: 22nd European Conference on Object-Oriented Programming*, July 2008.
20. H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11*, pages 297–306, 2003.
21. H. Rajan and K. Sullivan. Need for instance level aspect language with rich pointcut language. In *SPLAT workshop*, mar 2003.
22. H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05*, pages 59–68, 2005.
23. H. Rajan and K. J. Sullivan. Unifying aspect- and object-oriented design. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2008.
24. S. P. Reiss. Connecting tools using message passing in the field environment. *IEEE Softw.*, 7(4):57–66, 1990.
25. T. Rho, G. Kniesel, and M. Appeltauer. Fine-grained generic aspects. In *FOAL*, 2006.
26. K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04*, pages 16–25, 2004.
27. M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05*, pages 653–656, 2005.
28. K. J. Sullivan, I. J. Kaletyz, and D. Notkin. Mediators in a radiation treatment planning environment. *IEEE Transactions on Software Engineering*, 22, 1996.
29. K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM TOSEM*, 1(3):229–68, Jul 1992.
30. T. Tourwé, J. Brichau, and K. Gybels. On the existence of the AOSD-evolution paradox. In *SPLAT workshop*, March 2003.
31. M. Wand, G. Kiczales, and C. Dutchyn. A semantics for advice and dynamic join points in aspect-oriented programming. *ACM Trans. Program. Lang. Syst.*, 26(5):890–910, 2004.
32. A. K. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, Nov 1994.

(T-ASSOCIATE)

$$\frac{\Pi \vdash e_0 : \textbf{exp}\ c_0 \qquad \Pi \vdash e_1 : \textbf{exp}\ c_1}{\Pi \vdash \texttt{associate}(e_0, e_1) : \textbf{exp}\ c_0}$$

(T-EVTYPE)

$$\frac{\mathit{isClass}(c) \qquad \forall i \in \{1..n\}, \mathit{isType}(t_i)}{\vdash c\ \texttt{evtype}\ p\ \{t_1\ \mathit{var}_1\texttt{;} \ldots t_n\ \mathit{var}_n\texttt{;}\} : \mathrm{OK}}$$

(T-REGISTER)

$$\frac{\Pi \vdash e : \textbf{exp}\ c}{\Pi \vdash \texttt{register}(e) : \textbf{exp}\ c}$$

(T-INVOKE)

$$\frac{\Pi \vdash e : \textbf{exp}\ (\textbf{thunk}\ c)}{\Pi \vdash \textbf{invoke}(e) : \textbf{exp}\ c}$$

(T-UNDER)

$$\frac{\Pi \vdash e : \textbf{exp}\ t}{\Pi \vdash \textbf{under}\ e : \textbf{exp}\ t}$$

(T-EVENT)

$$\frac{\begin{array}{c}(c\ \textbf{evtype}\ p\ \{t_1\ \mathit{var}_1\texttt{;} \ldots t_n\ \mathit{var}_n\texttt{;}\}) \in CT \\ \{\mathit{var}_1 : \textbf{var}\ t_1, \ldots, \mathit{var}_n : \textbf{var}\ t_n\} \subseteq \Pi \qquad \Pi \vdash e : \textbf{exp}\ c' \qquad c' \preccurlyeq c\end{array}}{\Pi \vdash \textbf{event}\ p\ \{e\} : \textbf{exp}\ c}$$

(T-BINDING)

$$\frac{\begin{array}{c}(c, c'\ m\ (t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\ \{e\}) = \mathit{findMeth}(c, m) \qquad \mathit{isClass}(c') \\ t_1 = \textbf{thunk}\ c' \qquad (\forall i \in \{2..n\} :: \mathit{isType}(t_i)) \qquad \{\mathit{var}_2 : \textbf{var}\ t_2, \ldots, \mathit{var}_n : \textbf{var}\ t_n\} \subseteq \pi\end{array}}{\Pi \vdash (\textbf{when}\ p\ \textbf{do}\ m) : \mathrm{OK\ in}\ c}$$

(T-PROGRAM)

$$\frac{\forall i \in \{1..n\}, \mathit{decl}_i : \mathrm{OK} \qquad \emptyset \vdash e : \textbf{exp}\ t}{\vdash \mathit{decl}_1 \ldots \mathit{decl}_n\ e : \mathrm{OK}}$$

(T-NEW)

$$\frac{\mathit{isClass}(c)}{\Pi \vdash \textbf{new}\ c() : \textbf{exp}\ c}$$

(T-GET)

$$\frac{\Pi \vdash e : \textbf{exp}\ c \qquad \mathit{fieldsOf}(c)(f) = t}{\Pi \vdash e.f : \textbf{exp}\ t}$$

(T-CLASS)

$$\frac{\begin{array}{c}\mathit{isClass}(d) \qquad \forall i \in \{1..n\}\ \mathit{isClass}(t_i) \qquad \forall i \in \{1..n\}\ f_i \notin \mathit{dom}(\mathit{fieldsOf}(d)) \\ \forall j \in \{1..m\}\ \mathit{meth}_j : \mathrm{OK\ in}\ c \qquad \forall k \in \{1..l\}\ \mathit{binding}_k : \mathrm{OK\ in}\ c\end{array}}{\vdash \textbf{class}\ c\ \textbf{extends}\ d\ \{t_i\ f_i;\ \mathit{meth}_j;\ \mathit{binding}_k\} : \mathrm{OK}}$$

(T-METHOD)

$$\frac{\begin{array}{c}\mathit{isClass}(t) \qquad \forall i \in \{1..n\}, \{\mathit{var}_i : t_i, \textbf{this} : c\} \vdash e : \textbf{exp}\ t' \\ t' \preccurlyeq t \qquad \mathit{override}(m, t_1 * t_2 * \ldots * t_n \mapsto t) \qquad CT(c) = \textbf{class}\ c\ \textbf{extends}\ d\ \{\ldots\}\end{array}}{\Pi \vdash t\ m(t_i\ \mathit{var}_i)\{e\} : \mathrm{OK\ in}\ c}$$

(T-ASSIGN)

$$\frac{\Pi \vdash e_0 : \textbf{exp}\ t_0 \qquad \Pi \vdash e_1 : \textbf{exp}\ t_1 \qquad t_1 \preccurlyeq t_0}{\Pi \vdash e_0 = e_1 : \textbf{exp}\ t_1}$$

(T-SEQUENCE)

$$\frac{\Pi \vdash e_1 : \textbf{exp}\ t_1 \qquad \Pi \vdash e_2 : \textbf{exp}\ t_2}{\Pi \vdash e_1; e_2 : \textbf{exp}\ t_2}$$

(T-SET)

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

(T-DEFINE)

$$\frac{\mathit{isType}(t) \qquad \Pi \vdash e_1 : \textbf{exp}\ t_1 \qquad \Pi \vdash e_2 : \textbf{exp}\ t_2 \qquad t_1 \preccurlyeq t \qquad \Pi' = \Pi \uplus \{\mathit{var} : \textbf{var}\ t\}}{\Pi \vdash t\ \mathit{var} = e_1; e_2 : \textbf{exp}\ t_2}$$

(T-CAST)

$$\frac{\mathit{isType}(t)}{\Pi \vdash \textbf{cast}\ t\ e : \textbf{exp}\ t}$$

(T-NULL)

$$\frac{\mathit{isClass}(c)}{\Pi \vdash \textbf{null} : \textbf{exp}\ c}$$

(T-LOC)

$$\frac{(\mathit{loc} : \textbf{var}\ t) \in \Pi}{\Pi \vdash \mathit{loc} : \textbf{var}\ t}$$

(T-UNDER)

$$\frac{\Pi \vdash e : \textbf{exp}\ t}{\Pi \vdash \textbf{under}\ e : \textbf{exp}\ t}$$

(T-CALL)

$$\frac{\begin{array}{c}\Pi \vdash e : \textbf{exp}\ c' \qquad (c, t\ m\ (t_1\ \mathit{var}_1, \ldots, t_n\ \mathit{var}_n)\ \{e\}) = \mathit{findMeth}(c, m) \\ c' \preccurlyeq c \qquad \Pi \vdash e : \textbf{exp}\ t' \qquad t' \preccurlyeq t \qquad \Pi \vdash e_i : \textbf{exp}\ t_i\ \forall\, i \in \{1..n\}\end{array}}{\Pi \vdash e.m(e_1, \ldots, e_n) : \textbf{exp}\ t}$$

Auxiliary Functions:

$$\begin{array}{l}\mathit{isClass}(t) = (\textbf{class}\ t \ldots) \in CT \\ \mathit{isThunkType}(t) = (t = \textbf{thunk}\ c \wedge \mathit{isClass}(c)) \\ \mathit{isType}(t) = \mathit{isClass}(t) \vee \mathit{isThunkType}(t)\end{array}$$

Fig. 7. Type-checking rules for Ptolemy-I based on [19]

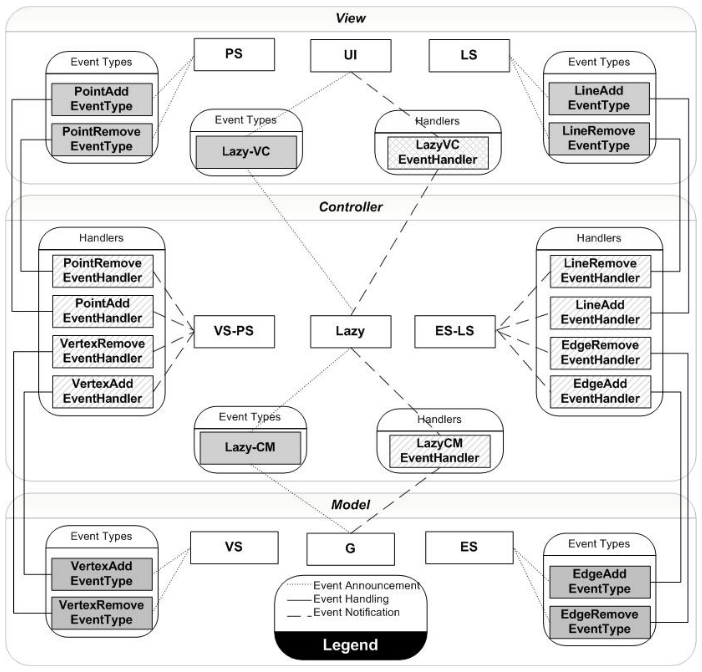

Fig. 8. Graph System Architectural Diagram Using Ptolemy-I