A Preliminary Study of Quantified, Typed Events

Robert Dyer[1] Mehdi Bagherzadeh[1] Hridesh Rajan[1] Yuanfang Cai[2]

[1]Computer Science - Iowa State University
{rdyer,mbagherz,hridesh}@cs.iastate.edu

[2]Computer Science - Drexel University
yfcai@cs.drexel.edu

Abstract

In previous work, Rajan and Leavens presented the design of Ptolemy, a language which incorporates the notion of quantified, typed events for improved separation of concerns. In this work, we present an empirical study to evaluate the effectiveness of Ptolemy's design by applying it to a series of architectural releases of a software product line (SPL) for handling multimedia on mobile devices, called MobileMedia, and the comparison and contrast of our findings with a previous in-depth analysis by Figueiredo et al of the object-oriented and aspect-oriented designs of the same system. Our comparative analysis using quantitative metrics proposed by Chidambar and Kemerer (and subsequently used by Garcia et al) and a net-options value analysis used earlier by Cai, Sullivan and Lopes shows that quantified, typed events significantly improve the separation of concerns and further decouple components in the MobileMedia design.

1. Introduction

An important class of separation of concerns techniques [5] allows programmers to modularize crosscutting concerns. Implicit invocation (II) languages and aspect-oriented (AO) languages in the style of pointcut-advice languages [18] are prominent examples in this class. *Quantified, typed events* [20] are also a modularization mechanism in this class of separation of concerns techniques.

The key idea behind quantified, typed events is to allow programmers to declare abstract events in the software system as *event types*. Often certain information about an abstract event is useful, thus these event type declarations may contain context information defined as one or more context variables. Certain components in the software system may raise these abstract events. This is done using *announce expressions*. An announce expression may name an event type p. The semantics of an announce expression naming an event type p is that when the program execution reaches the event expression, an event of type p is said to be raised. This is the key distinction from aspect-oriented languages [15], which generally do not require events to be explicitly raised.

Certain other components in the system may express an interest in being notified when one or more abstract events are raised in the system. This is done using *binding declarations*. A binding declaration may also name an event type p. The semantics of a binding declaration naming an event type p is to express an interest in *all raised events of type* p, without having to name the components that are responsible for raising those events. This is the key distinction from implicit-invocation languages [17], which do not directly provide the ability to quantify over all components raising an event without explicitly naming them.

The benefit of quantified, typed events over II languages is that observer methods are decoupled from the code that announces events; instead they only name event types. The benefit over AO languages is that advice can uniformly access reflective information about the join point while preserving its encapsulation, thus it is decoupled from the base code structure and the names used. In II languages, handlers (observers) are name dependent on subjects that announce events. AO languages remove this dependence but the implicit dependence remains [20]. In Ptolemy, event types (**`event`** declarations in syntax) decouple subjects that announce events from observers that register with these events via a well-defined interface [20].

In this work we rigorously analyze the software engineering benefits of Ptolemy. We present a detailed case study to evaluate the effectiveness of its design by applying it to a series of architecture releases of a software product line (SPL) for handling multimedia on mobile devices, called MobileMedia [8] and the comparison and contrast of our findings with a previous in-depth analysis [8].

2. An Example in Ptolemy

To illustrate the key ideas in Ptolemy [20] let us consider the class `LinkedList` shown in Figure 1. It maintains the linked list data structure using a nested class `Node` and provides a method `add` to insert elements in the list.

An example requirement for such a collection could be to declare and announce addition and removal of elements

ESCOT '10 Rennes and Saint-Malo, France.

```
1 void event ElementAdded { int element; }

3 class LinkedList {
4  Node h, t; /* Class Node elided */
5  void add(int num) {
6   announce ElementAdded(num) {
7    if (h == null) { h = new Node(num); t = h; }
8    else { t.next = new Node(num); t = t.next; }
9   } } }
10 class Average {
11  long count = 0; double currAverage = 0;
12  Average() { register(this); }
13  void update(thunk void rest, int element) {
14   invoke(rest);
15   currAverage=((currAverage*count) + element)/(++count);
16  }
17  when ElementAdded do update;
18 }
```

Figure 1. List and incremental averaging in Ptolemy

from the collection. Other components may be interested in such events, e.g. for implementing incremental functionality that relies on analyzing the increments. An example of such a concern for the linked list of integers is the requirement to keep track of the average of the integers in the list. Such an average would need to be updated when new integers are added and removed from the list. Furthermore, such functionality may not be useful for all applications that use the linked list class, thus it would be sensible to keep its implementation separate from that of the linked list class in order to maximize reuse of the linked list class.

Typically such a requirement would be implemented using the observer design pattern [10], that is directly supported in implicit invocation (II) languages [17]. In II languages, *events* are seen as a decoupling mechanism that is used to interface two sets of modules, so that they can be independent of each other. Certain modules, often called subjects, dynamically and explicitly *announce* events. Another set of modules, often called 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. Aspect-oriented (AO) languages [15] such as AspectJ [14] can also be used to implement this requirement. However, both II and AO languages have several limitations as described in [20].

In II languages, observers remain coupled with subjects, no support for overriding is available, and specifying how each event is handled can grow in proportion to the number of objects from which implicit invocations are to be received [20]. AO languages have a fragile pointcut problem [22], language-imposed limits on the types of events announced, and a limited interface for accessing contextual (or reflective) information about an event [20]. Quantified, typed events in Ptolemy are designed to solve these problems [20].

In Ptolemy, **event** declarations allow programmers to declare named event types. An event type declaration p has a return type, a name, and zero or more context variable declarations. These context declarations specify the types and names of reflective information communicated between announcement of events of type p and handler methods. These declarations are independent from the modules that announce or handle these events. The event types thus provide an interface that completely decouples subjects and observers. An example event type declaration is shown on line 1 in Figure 1. The **event** `ElementAdded` declares that an event of this type makes one piece of context available: the item (`element`) that is being inserted in the list.

Events are explicitly announced using **announce** statements. These expressions enclose a body, which can be replaced by a handler. This functionality is akin to **around** advice in AO languages. The class `LinkedList` declares and announces an event of type `ElementAdded` using an announce statement (lines 6–9). Arbitrary blocks can be declared as the body of the announce statement, which requires the context variables to be bound in the lexical scope of their declaration. Event type `ElementAdded` declares one context variable. Thus the **announce** statement binds `num` to the name `element` (line 6).

Finally, the names of **event** declarations can be utilized for quantification, which simplifies binding and avoids coupling observers with subjects. Bindings in Ptolemy associate a handler method to a set of events identified by an event type. The binding in Figure 1, line 17 says to run method `update` when events of type `ElementAdded` are announced. This allows selecting a number of event statements with just one succinct binding declaration without depending on the modules that announce events.

Each handler method in Ptolemy takes an event closure as the first actual argument. An *event closure* [20, 21] contains code needed to run the applicable handlers and the original event's code. An event closure is run by an **invoke** expression. The **invoke** expression in the implementation of the handler method `update` in Figure 1, line 14, causes all applicable handlers and the original event's code to run before computing the average incrementally.

3. Case Study: MobileMedia Application

To evaluate the software engineering benefits of Ptolemy, we applied the language design in the context of an existing software product-line application called MobileMedia [8]. This section introduces MobileMedia, details observations from our implementation of MobileMedia in Ptolemy and gives a quantitative analysis of the design in terms of standard software engineering metrics and a net options value analysis.

3.1 MobileMedia Software Product Line Overview

MobileMedia [8] is an extension of the MobilePhoto [25] application, which was developed to study the effect of aspect-oriented designs on software product lines (SPL). MobileMedia is an SPL for applications that manipulate photos, music and videos on mobile devices.

MobileMedia extends MobilePhoto to add new mandatory, optional and alternative features. There are a total of 8

releases and descriptions of each is shown in Figure 2. For example in release 7 (R7) a new feature is added to manage music and an optional feature added in a previous release to manage photos is turned into an alternative feature.

Release	Description	Type of Change
R1	MobilePhoto core	
R2	Exception handling included (in the AspectJ and Ptolemy versions, exception handling was implemented according to [9])	Inclusion of non-functional concern
R3	New feature added to count the number of times a photo has been viewed and sorting photos by highest viewing frequency. New feature added to edit the photo's label	Inclusion of optional and mandatory features
R4	New feature added to allow users to specify and view their favourite photos.	Inclusion of optional feature
R5	New feature added to allow users to keep multiple copies of photos	Inclusion of optional feature
R6	New feature added to send photo to other users by SMS	Inclusion of optional feature
R7	New feature added to store, play, and organise music. The management of photo (e.g. create, delete and label) was turned into an alternative feature. All extended functionalities (e.g. sorting, favourites and SMS transfer) were also provided	Changing of one mandatory feature into two alternatives
R8	New feature added to manage videos	Inclusion of alternative feature

Figure 2. Summary of Change Scenarios in the MobileMedia SPL (based on Figueiredo *et al.*'s work [8, Tab.1])

For this case study, we implemented the 7 changed releases (R2–R8) using Ptolemy. For each release we started with the AO version and used it as a template for the Ptolemy release, creating one handler class for each aspect in the system. For each advice body in the aspect, a new handler method was added to the handler class. Event types were created and event announcement added to emulate AspectJ's pointcut-advice semantics.

An overview of the MobileMedia architecture in Ptolemy is shown in Figure 3. For each component, the release it first appears in is marked with a $+$R notation. If the component is changed in any release it is marked with a $\sim$R notation. For example, $MusicAccessor$ class in model was added in R7 and is marked in Figure 3 with a $+$R7 while $PhotoAccessor$ was changed in the same release as it is marked with a $\sim$R7.

Since we used the AO releases as a template for the Ptolemy releases, the architecture is similar to the architecture for the AO releases [8, Fig.4]. The key difference is that aspects are represented in our figure as classes and we have additional components shown for the event types (shaded). By adding quantified, typed events we made the implicit coupling of the aspects to the base components explicit and provided an interface between them.

3.2 Key Observations

We used the AO version of MobileMedia as a guide to creating the Ptolemy version. This allowed us to gain some interesting insights into the benefits of using each language, which we discuss in this section.

3.2.1 Observed Benefits of Aspect-oriented Design

The observed benefits of using aspect-oriented approaches come from the static crosscutting features of AspectJ, such as inter-type declarations and declare statements.

Inter-Type Declarations. A static feature of AspectJ that allows adding fields/methods to other classes is inter-type declarations (ITDs) [12, 14]. This feature allowed modular extensions of existing classes for the AO versions, but was not available for the Ptolemy versions (although this would be a reasonable extension to include in the future).

For the Ptolemy versions, ITDs were emulated as follows. For any field f introduced into a class C, a hashtable was added to the event handler. Getters and setters were generated that take an additional argument of type C, which was used as a key in the hashtable. For any methods introduced, a similar pattern was applied by placing the methods in the event handler and adding an additional parameter of type C.

In certain cases this required either increasing the visibility of fields in C or adding getters/setters for the field (which would typically be done by declaring the aspect as privileged in AspectJ). This pattern occurred frequently for releases 7 and 8 and as will be shown in Section 3.5 had a negative impact on our design.

Declare Parents. Similar to ITDs, type hierarchies in the base components can be extended in a modular manner using AspectJ's *declare parents* [12]. In the case of MobileMedia, two type hierarchies were extended by adding a new super-class to two base components. The effects of these extensions are easily modeled as inter-type declarations and handled in the Ptolemy version as previously described.

Softened Exceptions. AspectJ also has the ability to soften exceptions thrown in the base components [14] using *declare soft* statements. This was used in MobileMedia to help modularize the exception handling feature in release 2, allowing the base components to no longer declare that it throws checked exceptions handled by the aspects.

The current implementation of Ptolemy does not have any similar constructs and thus the base components must still declare that these checked exceptions are thrown. There are both pros and cons of these declarations. The con is that a programmer must write these additional annotations. On the positive side, having these annotations makes the features in Ptolemy completely (un)pluggable [12]. In the AO version, if a feature that relies on softening exceptions is unplugged the compilation of the base components will fail.

3.2.2 Observed Benefits of Quantified, Typed Events

Quantification Failure. Ptolemy gives the programmer the ability to add event announcement for any arbitrary statement in the base components. AspectJ can only advise join points available in the provided pointcut language, such as method executions or calls. This often results in what Sullivan *et al.* have called quantification failure [23, pp.170]

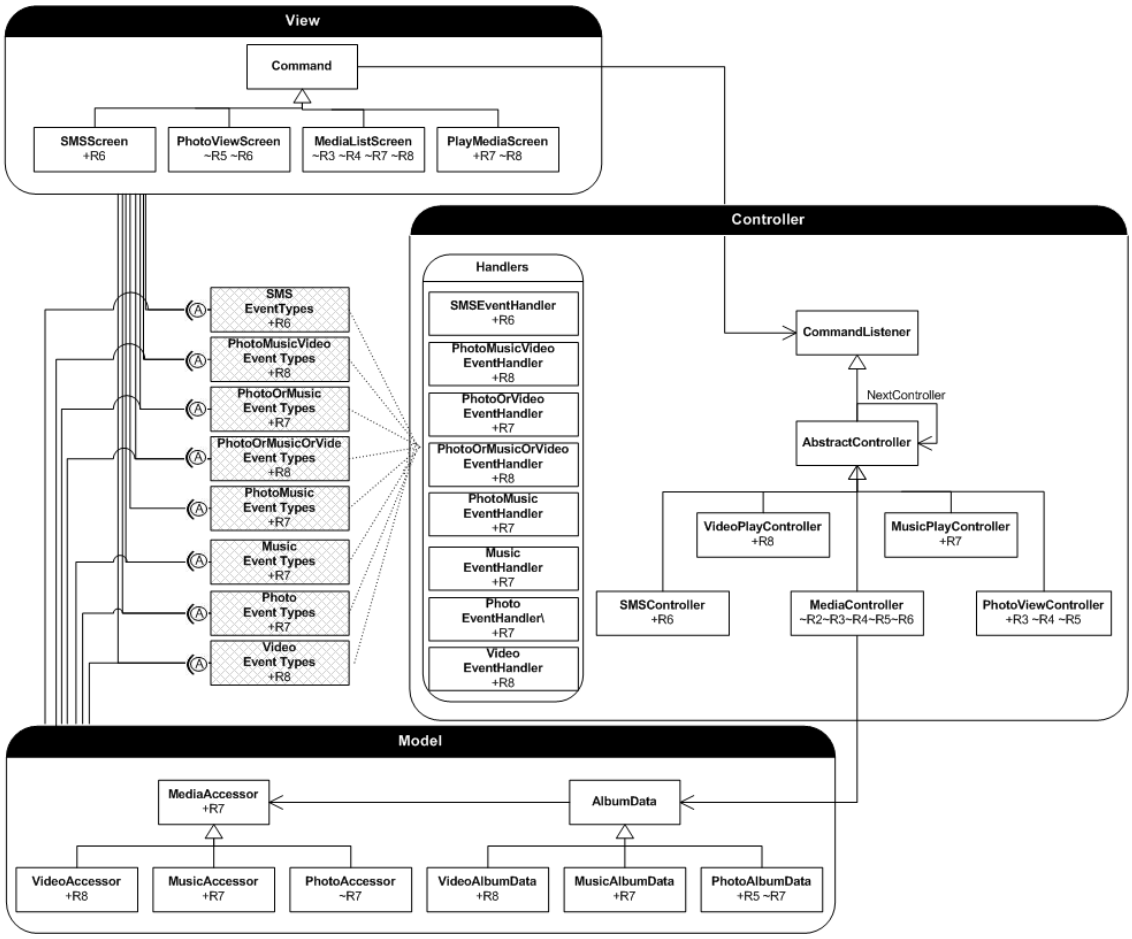


Figure 3. Ptolemy MobileMedia Architecture.

and is caused by incompleteness in the language's event model. Quantification failure occurs when the event model does not implicitly announce some kinds of events and hence does not provide pointcut definitions that select such events [23, pp.170]. In AspectJ-like AO languages there is a fixed classification of potential event kinds and a corresponding fixed set of pointcut definitions. For example some language features, such as loops or certain expressions, are not announced as events in AspectJ and have no corresponding pointcut definitions [23, pp.170].

In the MobileMedia application, we observed several instances of quantification failure. For example, in R2 the aspects needed to advise a *while* loop and similarly in R3 the aspects needed to advise a *for* loop. To accomodate this, the AO version refactors the base components, for example moving these loops into newly added methods. The Ptolemy version of MobileMedia did not suffer from this problem and thus these refactorings were not necessary.

Fragile Pointcut Problem. As mentioned by Figueiredo *et al.* [8], the AO version of MobileMedia suffers from a fragile pointcut problem [8,20,23]. This could be easily observed in release 7, where a mandatory feature PHOTO is generalized into two alternative features PHOTO or MUSIC. This generalization required modifying many pointcuts, previously relying on an implicit matching of signatures in the base.

The renaming of the base components itself is not a problem in the Ptolemy release and in fact requires no modification of events or handlers; the handlers will match on the event type which is not changed. If the event type is renamed (for example, to remain consistently named to the base components) then all handlers and events for that event type must be updated accordingly. The key difference in these two scenarios is that in the AO case, the developer must be aware of which pointcuts matched the given join point (which can be aided with tools such as AJDT) while in the case of Ptolemy, the compiler will specify type errors for every handler and publisher for that event type. This avoids the fragile pointcut problem entirely.

Advising Advice. As Rajan and Sullivan observed, the current implementation of AspectJ considers each advice body anonymous and thus they are not individually available for selection in pointcuts [21]. The pointcut designator *adviceexecution* is available in the language, however it selects every advice in the system. This can be narrowed to all ad-

vice in a single aspect, but not to a single, specific advice body. For MobileMedia this led to a problem for the AO version maintaining modularized exception handling in later revisions. Some advice bodies added in later revisions contain exception handling which could not be modularized. For Ptolemy this problem does not exist due to the ability to announce events inside event handlers. The newly added event handling code representing the advice body can simply announce an event, which will be handled by the exception event handlers.

3.3 Effect on Change Propagation

A key benefit of a modular software design is in its ability to hide design decisions that are likely to change [19]. Thus, we consider the number of changed components as a result of a change in a design decision to be an important comparator for a software design. To quantify this, similar to Figueiredo *et al.* [8], we measured the number of added and removed components in the system for each version as well as the number of components changed. The results are shown in Figure 4 with additional Ptolemy-specific (PTL) results for the number of event types added, changed or removed. For comparison we also include the AO-specific number of pointcuts (PCs) added, changed or removed.

For all releases, new components added to the AO version are also added to the Ptolemy version. The impact on the analysis for these components is approximately the same for both AO and PTL versions. The remaining differences in the number of added components is due to our refactoring of the command pattern in the base components to use quantified, typed events.

			R2	R3	R4	R5	R6	R7	R8
Components	Added	OO	9	1	0	5	7	17	6
		AO	12	2	3	6	8	21	16
		PTL	13	4	2	6	8	23	18
	Removed	OO	0	0	0	0	0	10	1
		AO	1	0	0	0	0	8	0
		PTL	1	1	0	1	0	7	2
	Changed	OO	5	8	5	8	6	12	22
		AO	5	10	2	8	5	16	9
		PTL	11	8	1	9	5	16	8
PCs	Added	AO	43	6	7	2	7	19	26
	Removed	AO	0	0	0	0	0	0	5
	Changed	AO	0	8	0	16	2	50	2
Event Types	Added	PTL	25	9	1	5	5	9	4
	Removed	PTL	0	1	0	1	0	3	0
	Changed	PTL	0	0	0	0	0	13	0

Figure 4. Change propagation in MobileMedia for each release (based on Figueiredo *et al.*'s work [8, Tab.3]).

In R7 a mandatory feature was turned into two alternative features, leading to changes in the base components which propagated to the event types and event handlers. 11 of the 13 resulting event type changes were due to the renaming of base components that were passed as context in those events types. Consider on the other hand the AO version which required changing 35 of the 50 pointcuts due to the fragility of the pointcuts.

In R8, several new alternate features were added to the system. The Ptolemy version was able to re-use several existing event types leading to the addition of only 4 new event types. The AO version however required adding 26 new pointcuts to the system.

Releases 7 and 8 both added alternate features to the system. The AO solutions for these revisions relied heavily on inter-type declarations where previous versions mostly relied on pointcuts and advice. Our method of emulating the ITDs required changing the aspect handler to insert hashtables and the introduced fields/methods. For R7 and R8 these changes occur in new handlers and thus do not affect the number of changed components.

In summary, for some versions quantified, typed events showed an improved ability to withstand change in components. In particular, for versions where significant refactoring in the base components took place, Ptolemy's design was able to reduce the impact of these changes in the base code from the handlers.

3.4 Effect on Coupling

As previously discussed, the main difference between AO languages and Ptolemy is that the dependency between components that announce events is explicitly stated using event statements that name event types. In AO languages this dependency is implicitly defined by the language semantics. Explictly naming event types introduces coupling. The aim of this section is to study the change in coupling between components. In order to evaluate this we used a subset of the metrics suite proposed by Chidambar and Kemerer [4] and Fenton and Pfleeger [7] and subsequently refined by Garcia *et al.* [11].

			R2	R3	R4	R5	R6	R7	R8
LCOO	Average	OO	6.38	10.08	10.88	8.83	10.24	12.04	11.80
		AO	5.67	8.69	8.50	6.97	8.24	9.39	8.03
		PTL	3.30	3.36	3.21	2.86	2.83	2.77	2.96
	Max	OO	64	70	71	73	96	113	114
		AO	64	94	94	64	85	109	111
		PTL	84	122	122	66	66	112	153
CBC	Average	OO	1.46	2.00	2.36	3.17	3.30	3.54	3.96
		AO	1.30	1.72	1.84	2.50	2.65	2.76	2.69
		PTL	0.80	0.94	1.06	1.43	1.57	1.71	2.01
	Max	OO	9	13	13	11	11	15	20
		AO	9	13	12	10	13	14	14
		PTL	9	14	14	12	13	14	14

Figure 5. Coupling and Cohesion for MobileMedia.

Lack of cohesion in operations (LCOO) and coupling between components (CBC) for each component of all 7 modified releases was measured for each design. Figure 5 shows the maximum value and the average across all components, while the minimum values were the same for each design and are thus omitted. The total coupling is higher for PTL versions, however the max CBC remains consistent with AO versions.

Figure 6 shows the number of components (NOC) and total lines of code (LOC) for each version. Since the implicit

coupling between pointcuts and base components in the AO versions was made explicit using event types in the Ptolemy version, the number of components for the Ptolemy version is now increased over the AO version by the number of event types in the system.

Note the number of attributes (NOA) is higher due to the explicit naming of context information but the number of operations (NOO) is lower than AO versions due to the lack of refactoring to expose join points.

		R2	R3	R4	R5	R6	R7	R8
LOC	OO	1159	1314	1363	1555	2051	2523	3016
	AO	1276	1494	1613	1834	2364	3068	3806
	PTL	1605	1923	2049	2374	2969	3655	4508
NOC	OO	24	25	25	30	37	46	51
	AO	27	29	32	38	46	59	75
	PTL	56	67	70	79	92	112	132
NOA	OO	62	71	74	75	106	132	165
	AO	62	72	76	77	111	139	177
	PTL	72	82	86	88	121	146	185
NOO	OO	124	140	143	160	200	239	271
	AO	158	187	199	230	285	345	441
	PTL	143	169	179	197	247	308	369

Figure 6. The measured size metrics for MobileMedia.

In summary, our results show the total coupling is slightly higher in Ptolemy versions due to increased separation of functionality into different components. This is the cost required for the improved change propogation gained by the Ptolemy design.

3.5 Net Options Value Analysis of Design

In this section, we use Baldwin and Clark's design rule theory and design structure matrix modeling to compare the AO and Ptolemy versions of the MobileMedia software product line in terms of their ability to accommodate mandatory, optional, and alternative features. This analysis builds on the prior work of Sullivan *et al.* [23, 24] and Lopes *et al.* [16], among others.

Design Rule (DR) Theory and Design Structure Matrix (DSM). Baldwin and Clark's theory is based on the idea that modularity adds value in the form of real options. An option provides the right to make an investment in the future, without a symmetric obligation to make that investment. Because an option can have a positive payoff but need never have a negative one, an option has a positive present value. Baldwin and Clark proposed that a module creates an option to invest in a search for a superior replacement and to replace the currently selected module with the best alternative discovered, or to keep the current one if it is still the best choice. Intuitively, the value of such an option is the value realized by the optimal experiment-and-replace policy.

Baldwin and Clark used *design structure matrices* (DSMs) [6] as the fundamental model of their option-based modularity theory. DSMs represent and depict, in matrix form, pair-wise dependencies between dimensions of a design space. The columns and rows of a DSM are labeled with design variables modeling design dimensions in which decisions are needed. A mark in a cell indicates that the decision on the row depends on the decision on the column.

Baldwin and Clark's theory defines a model for reasoning about the value added to a base system by modularity. This model states that splitting a design into m modules increases its base value S_0 by a fraction obtained by summing the *net option values* (NOV) (NOV_i) of the resulting options. NOV is the expected payoff of exercising a search and substitute option optimally, accounting for both the benefits and cost of exercising options. The value of a software with m modules is calculated as:

$$V = S_0 + NOV_1 + NOV_i + ... + NOV_m, \text{ where}$$
$$NOV_i = max_{k_i}\{\sigma_i n_i^{1/2} Q(k_i) - C_i(n_i)k_i - Z_i\}$$

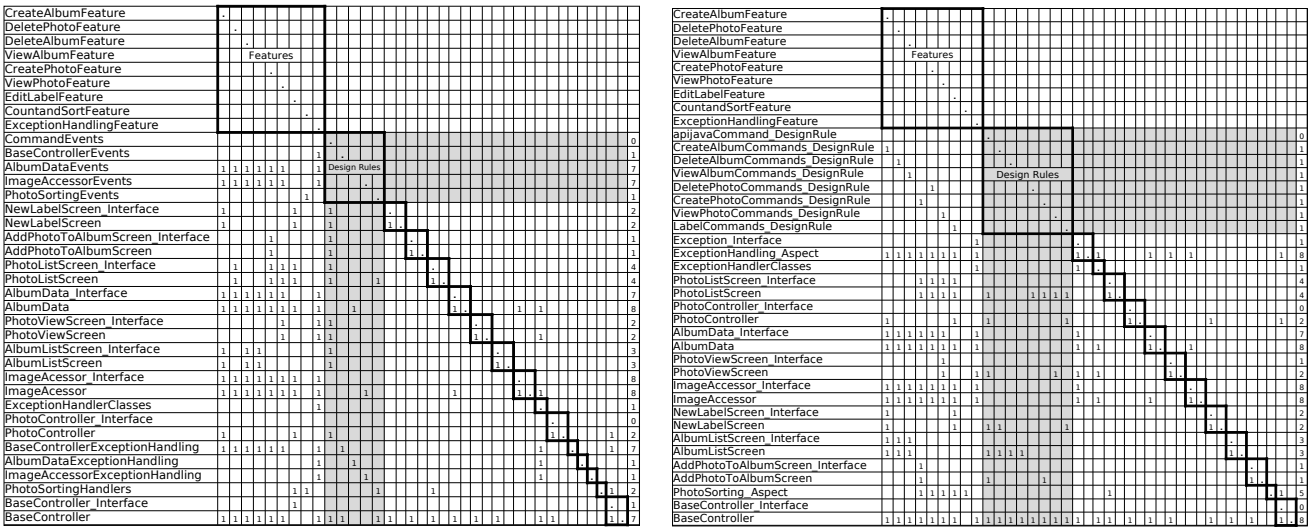
For module i, $\sigma_i n_i^{1/2} Q(k_i)$ is the expected benefit to be gained by accepting the best positive-valued candidate generated by k_i independent experiments. $C_i(n_i)k_i$ is the cost to run k_i experiments as a function C_i of the module complexity n_i. $Z_i = \Sigma_{j \text{seesi}} cn_j$ is the cost of changing the modules that depend on module i. The max picks the experiment that maximizes the gain for module i.

Our NOV Analysis Approach. In order to apply NOV analysis to compare the modularity differences between OO, AspectJ, and Ptolemy, we generated DSMs for each of the eight releases of MobileMedia using each paradigm and calculated the NOV values based on these 24 DSMs. Instead of generating DSMs directly from source code using reverse engineering tools, we use DSMs generated from UML component diagrams for a number of reasons.

First, as a new programming language, there is no static analysis tool that can faithfully extract dependency relations from Ptolemy. Second, there are a number of implicit but critical design decisions in OO an AspectJ versions that are not extractable from the source code. Last and the most important, using component diagrams allows us to compute one critical parameter easily, the technical potential, which measures how likely a module is subject to change.

Following the previous work of Sullivan et al. [23,24] and Cai et al. [1, 2, 13], we relate *environmental variables* with its technical potential. The rationale is that: the more environmental conditions influence a module, the more likely it is going to change. In this paper, we use features as environmental variables. That is, for any component, the more features influence it, the more technical potential it has. Using component diagrams allow us to assess which components are influenced by which feature easily and uniformly. To avoid mistakes caused by manually generating DSMs, we automatically translate a component diagram into an *augmented constraint network* [1–3], from which a DSM can be automatically generated.

Our Assumptions. We make the following assumptions in order to calculate NOV values and to make comparisons.



(A) DSM for Ptolemy's Design

(B) DSM for AO Design

Figure 7. DSM Models for MobileMedia Release 3

Following Baldwin and Clark, we use the number of variables to represent complexity. The *complexity* of a module can be measured as the size of the module as a proportion of the overall system. For example, in Figure 7 that shows the MobileMedia DSMs automatically generated from their component diagrams, there are 22 design variables shown in the lower right corner. Each component is modeled using two variables, the interface variable and the implementation variable. We thus model the complexity of the whole system as 1, model the complexity of a component as 2/22, and model the complexity of an aspect module as 1/22. The number of modules are the number of blocks of diagonal in a DSM, that is, the number of components, aspects, and event handlers. We assume that the cost of each experiment on a module with unit complexity is 1.

The *visibility cost* measures the cost incurred by dependencies between modules. From the automatically derived DSM model, for any module, we can automatically calculate which and how many other variables depend on it. All the ripple dependencies are already taken into account by the underlying constraint network. As a result, we model the *visibility cost* of a module as the sum of the cost of redesigning each dependent module, calculated by multiplying the complexity and unit cost of each dependent module.

The most important parameter for NOV analysis is *technical potential*, σ. Technical potential is the expected variance on the rate of return on an investment in producing variants of a module implementation. On the assumption that the prevailing implementation of a module is adequate, the expected variance in the results of independent experiments is proportional to changes in requirements that drive the evolution of the module's specification. We relate this estimation with the number of features that influence the module. For example, if all features influence a module, then this module is highly likely to change.

	R1	R2	R3	R4	R5	R6	R7	R8
OO	0.77299	0.77299	0.77299	0.77299	1.62165	2.05698	2.67292	3.94591
AO	0.77299	0.94178	0.78183	0.82255	1.83999	2.50879	3.13572	4.78385
PTL	0.77299	1.19178	0.94179	0.89245	1.86611	2.58423	2.82729	3.83676

Figure 8. Net option values for the three designs: note the high values for R7 and R8, which show the usefulness of ITDs in AO languages compared to workarounds in Ptolemy.

Results. Given the above assumptions, we calculate the NOV values for each release and each paradigm. These are shown in Figure 8. It is important to put these results in the perspective of language features most valuable for each of these releases. From R2 to R6, when mandatory and optional features are added, most used features in the AO design were pointcuts and advice. In R7 and R8, when alternative features are added, most used language features in the AO design were ITDs. In R2 to R6, in Ptolemy's design, pointcut-advice were replaced by quantified, typed events. In R7 and R8, we extensively worked around the lack of inter-type declarations in the Ptolemy designs.

From the results, it is clear that for R2 to R6 the Ptolemy designs generate higher NOV values compared to the AO designs. This is because in the Ptolemy designs, there are less dependencies between base code and the event handler, which allows for their independent evolution. In R7 and R8, the AO designs generate higher NOV values compared to the Ptolemy designs. The primary reason for this result is that in the Ptolemy designs, due to the workaround to emulate ITDs, the dependencies increase significantly between source code and event handlers. It is clear from the result of R7 and R8 that ITDs are a beneficial feature and that our workaround in the Ptolemy designs is not satisfactory. These

results show that the quantified, typed events as supported by Ptolemy are more suitable for mandatory and optional features. It would be interesting to explore the NOV values for a language design that supports both quantified, typed events and inter-type declarations.

4. Conclusion and Future Work

Finding a good separation of concerns is an important problem [5]. It is vital for improving the reliability and evolution of software systems. New modularization mechanisms enable improved separation of concerns. Their invention and refinement is thus equally important for maintaining intellectual control on the growing complexity of software systems [5]. Quantified, typed events [20] is one such modularization mechanism.

In this paper, we presented a rigorous evaluation of quantified, typed events on an already well-substantiated case study [8]. The results of our analysis using standard design metrics [4, 7, 11] show that even though quantified, typed events make coupling between components explicit, the overall coupling and cohesion of the system does not increase significantly (Section 3.4). We also found several instances of the quantification failure and fragile pointcut problem, where Ptolemy's use of quantified, typed events helped reduce the impact of change (Section 3.3). The results of our net-options value analysis [1, 2, 13, 16, 23, 24] demonstrate that the lack of AO-style inter-type declarations degrades the separation of concerns in Ptolemy. However, quantified, typed events improve the separation of concerns for mandatory and optional features thereby increasing the value of the software design (Section 3.5).

Acknowledgments. Authors were supported in part by the NSF grants CNS 06-27354 and CNS 08-08913. Comments from Youssef Hanna and Yuly Suvorov were helpful.

References

[1] Y. Cai, S. Huynh, and T. Xie. A framework and tools support for testing modularity of software design. In *ASE*, 2007.

[2] Y. Cai and K. Sullivan. Modularity analysis of logical design models. In *ASE'06*.

[3] Y. Cai and K. Sullivan. Simon: A tool for logical design space modeling and analysis. In *ASE*, 2005.

[4] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE TSE*, 20(6):476–493, 1994.

[5] E. W. Dijkstra. On the role of scientific thought. *EWD 477*, August 1974.

[6] S. D. Eppinger. Model-based approaches to managing concurrent engineering. *Journal of Engr Design*, 2(4):283–290, 1991.

[7] N. E. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. 1998.

[8] E. Figueiredo, N. Cacho, C. Sant'Anna, M. Monteiro, U. Kulesza, A. Garcia, S. Soares, F. Ferrari, S. Khan, F. Castor Filho, and F. Dantas. Evolving software product lines with aspects: an empirical study on design stability. In *ICSE'08*.

[9] F. C. Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira. Exceptions and aspects: The devil is in the details. In *FSE*, 2006.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. 95.

[11] 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*, pages 3–14, 2005.

[12] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA '02*, pages 161–173.

[13] S. Huynh, Y. Cai, Y. Song, and K. Sullivan. Automatic modularity conformance checking. In *ICSE'08*.

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

[15] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP 97*.

[16] C. V. Lopes and S. K. Bajracharya. An analaysis of modularity in aspect-oriented design. In *AOSD'05*.

[17] D. C. Luckham and J. Vera. An event-based architecture definition language. *TOSEM*, 21(9):717–734, 1995.

[18] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In *ECOOP '03*, pages 2–28.

[19] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–8, December 1972.

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

[21] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE 2005,pp. 59–68*.

[22] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM '05*.

[23] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *ESEC/FSE 2005, pp. 166–175*, 2005.

[24] K. J. Sullivan, W. G. Griswold, Y. Cai, and B. Hallen. The structure and value of modularity in software design. In *ESEC/FSE'01*.

[25] T. Young. Using AspectJ to build a software product line for mobile devices. Master's thesis, University of British Columbia, 2005.