Eos: Instance-Level Aspects for Integrated System Design

Hridesh Rajan
Dept. of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296
hr2j@cs.virginia.edu

Kevin Sullivan
Dept. of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2206
sullivan@cs.virginia.edu

ABSTRACT

This paper makes two contributions: a generalization of *AspectJ*-like languages with first-class aspect instances and instance-level advising, and a mapping of the mediator style for integrated system design into this space. We present *Eos* as a prototype language design and implementation. It extends *C#* with *AspectJ*-like constructs, first-class aspect instances and instance-level advising. These features enable a direct mapping of mediators to aspect instances, with modularity improved, insofar as components need not declare, announce, or register for events.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *first-class aspect instances, instance-level advising;* D.2.2 [**Software Engineering**]: Design Tools and Techniques – *mediator design;* D.2.11 [**Software Engineering**]: Software Architectures – *implicit invocation, patterns, mediator*

General Terms

Design, Experimentation, Languages.

Keywords

Design, mediators, integration, aspects, C#, instances

1. INTRODUCTION

Component integration creates value by automating the costly and error-prone task of imposing desired behavioral relationships on components manually. A problem is that straightforward software design techniques map integration requirements to scattered and tangled code, compromising modularity in ways that dramatically increase development and maintenance costs.

Sullivan and Notkin devised the mediator design approach to address this problem [31][32][34]. It enables and promotes the modular representation of behavioral relationships. By a *behavioral relationship* we mean a protocol for coordinating the control, actions, and states of subsets of system components to satisfy part of the integration requirements for the system.

The mediator style maps each kind of behavioral relationship in a system to a corresponding, object-oriented mediator class. An instance of such a class represents an instance of the relationship that integrates particular instances of subject classes. Mediators require that the objects to be integrated announce declared events to signal actions or changes that the mediators have to handle.

Recent aspect-oriented (AO) [22] methods similarly seek modular representation of requirements that otherwise map to tangled and scattered code, and so to poorly modularized and unnecessarily costly designs. An aspect in an AO language is a modular representation of a *crosscutting concern*, while a mediator is a modular representation of a behavioral relationship, which can be seen as a particular kind of crosscutting concern.

AO methods thus suggest a both critique of, and an improvement on, the mediator style. The mediator approach demands explicit registration with explicitly declared and announced events. AO languages, by contrast, provide join points as implicit, language-defined events, and pointcuts, which enable implicit registration with quantified subsets of join points.

The critique is that mediators do not fully modularize behavioral relationships, for two reasons. First, they impose constraints on the components to be integrated—that they must expose events matching the needs of mediators—thus components classes might have to change to accommodate new mediators. Second, a mediator integrating a quantified set of components will have to be changed to register with different events if that set changes.

The suggested enhancement is to use aspects as mediators, with join points and pointcuts instead of explicit events. Because AO components implicitly expose join points as events, no explicit declarations are needed. Because pointcuts are predicates on join points, changes in registration can occur automatically.

In an earlier paper [33], we reported our experience mapping the mediator approach into the design space of *AspectJ* [8]. The results were encouraging but mixed and led to a corresponding critique of the *AspectJ* design. The language doesn't provide first-class aspect *instances* or *instance-level advising*, by which we mean the instantiation of aspects using *new*, and selective advising of the join points of individual object *instances*. Rather, the model is one of aspects as constructs that modify classes, thus all instances of a given class. Work-arounds are possible, but incur unnecessary performance or design costs.

ESEC/FSE'03, September 1–9, 2003, Helsinki, Finland.

The problem that we address in this paper is our inability to map mediators to aspects in a completely satisfactory way. We make two contributions: an innovative extension of such languages with first-class aspect instances and instance-level advising, with *Eos* as a functioning prototype; and, second, proof that the resulting model supports a fully fledged, aspect-oriented variant of mediator-based design that relieves developers of the need for explicit event declaration, announcement, and registration.

The rest of this paper is as follows. Section 2 reviews mediator-based design. Section 3 reviews aspect-oriented programming. Section 4 discusses mediators as aspects, and the work-arounds needed today. Section 5 presents *Eos,* our new language design and implementation, and the mapping of mediators to aspects in *Eos.* Section 6 presents an evaluation of this research. Section 7 summarizes and discusses future work.

2. INTEGRATION BY MEDIATORS

Consider a system with instances, *m,* of class *model*, and *v,* of class *view*, and a requirement that *m* and *v* remain consistent. The requirement calls for a behavioral relationship, *update(m,v),* on *m* and *v*: *without recursion, if* v *changes, update* m*, and if* m *changes, update* v. Multiple *views*, *v1 and v2,* imply multiple relationships, *update(m,v1), update(m,v2).* An integrated system is seen as a set of component behaviors integrated in a network of behavioral relationships. In this case, changing a *view* would cause a *model* update, and then an update to the second *view.*

2.1 Behavioral Relationships

The example reflects a key step in mediator design: separating behavioral relationships and component behaviors. Many people, apparently unaware of the possibility, structure systems as sets of components that interact directly and in complex ways.

If one makes the separation, the next question is how best to map behavioral relationships to code. Many straightforward design methods map relationships—whether separated in design concept or not—to non-modular structures. Behavioral relationships end up mapped not to their own modules but to code and data scattered among and intermingled with modules implementing component behaviors, complicating and coupling them.

For example, *view* classes are often designed not only to present data, but to implement *model-view* relationships, which involve data conversions and calls to *model* objects to set and get their state. Separable *view* and *update* concerns are mapped to a *view* class, merging concerns and coupling *view* to *model* by procedure calls. Individual components and overall systems can quickly become unmanageable as the level of integration increases.

2.2 Mediators

The mediator approach averts this problem. The approach has three parts. One structures an integrated system as a collection of component behaviors integrated by behavioral relationships. One then maps components and relationships to instances of corresponding classes, with mediators as modular representations of behavioral relationships. Finally, one uses implicit invocation (event notification) [15] to create required *invokes* relations from components to mediators without inducing *names* dependences.

In this style, in addition to providing *methods* that can be called, components declare and announce *events*. Other components—mediators—can register operations to be invoked by events. The rationale for this approach is that visible actions of a component are part of its interface, and interfaces should be explicit.

A mediator design of our model-view system would include classes *model, view* and *update* (a mediator class), with instances *m, v1, v2, u1, and u2.* The *model* and *view* interfaces would provide methods *setState* and *getState* and an event, *changed,* announcing changes. (We elide parameters for presentation.) Each instance of *update*, say *u1,* would store references to *m*, and to a *view*, *v1.* It would register *u1.viewUpdate(…)* with *m.changed(…)* and *u1.modelUpdate(…)* with *v1.changed(…).* When invoked by *v1.changed(…)*, *u1.modelUpdate* would return immediately on a recursive call, and otherwise convert the event parameters and call *v1.setState(…).* The *u1.viewUpdate* method would work similarly in the opposite direction.

The *model* and *view* components remain uncomplicated by integration code and are *name*-independent of each other and of the mediator class, and thus also independent as compilation and link units. Meanwhile, the mediator classes modularize the code and data and *invokes* and *names* relations implementing the relationships. Systems designed this way are modular and easier to develop and maintain than straightforward designs.

3. ASPECT-ORIENTED PROGRAMMING

In the decade since the mediator approach was devised, the idea of aspect-oriented programming (AOP) has emerged [22]. The mediator approach and AOP are related in important ways. First, they both address the problem that traditional methods map key requirements to non-modular designs. The mediator approach addresses integration requirements, in particular. AOP addresses a broader range of requirements. Second, both methods attack the problem with novel modular program constructs and methods for using them. In particular, both techniques work in large part by enabling new kinds of modules to arrange for their methods to be invoked implicitly by execution events of other objects.

A crucial difference between the approaches is in the underlying event models. The mediator approach assumes that events are explicitly declared, components explicitly announce events, and mediators explicitly register with events. The aspect-oriented model, by contrast, assumes that so many events—and subsets of events—are of potential interest that it is unreasonable to name them explicitly. AOP languages thus make classes of execution events visible as *join points* in the language semantics, and they provide *pointcut* expressions—predicates on the join points of a program—as a means of registering methods with sets of events.

4. MEDIATORS AS ASPECTS

The aspect perspective suggests a critique of the mediator style, a variant based on aspects, and a subsequent critique of current aspect language designs. First, mediators impose constraints on components—that they declare and announce events. It can thus be argued that mediators do not fully modularize behavioral relationships. Components have to expose events matched to the needs of mediators. Thus, adding a new behavioral relationship and corresponding mediator class can require changes to multiple component classes—in the worst case, across a whole system.

Accepting this critique leads to an idea for improving mediators. The idea is to implement them as aspects, using join points and pointcuts in place of explicit events. Both styles modularize behavioral relationships. Both preserve the *name* independence of the components being integrated by providing a mechanism that enables one component to *invoke* another without *naming* it. However, mediators do this by using explicit events (implicit invocation) while aspects use implicit events (join points).

This difference focuses on a distinguishing property of AOP: it rejects an *explicit interface principle* for events. The rationale rests on several crucial propositions. First, it is too hard for designers to anticipate the events that components might need to observe. Second, it is costly and error-prone to have to explicitly declare, announce, and register with events. Third, it is unnecessary to make events explicit; rather, a language can make broad classes of events visible as join points. Fourth, the costs of abandoning narrow, explicit event interfaces are less than the benefits of broad, implicit event interfaces.

The validity of these propositions is disputable. There are real benefits to explicit interfaces; nor do we have aspect-oriented languages that make all potentially interesting execution events visible as join points: It hard for a language designer to anticipate all the events of potential interest, too. The explicit approach has the benefit of enabling announcement of any declared event whatsoever.

Moreover, the underlying AO assumptions might be valid for some systems and not for others. In any case, their validity is an unresolved empirical question, and is beyond the scope of this work. We simply stipulate that they are valid, believing that there are important classes of systems for which that is the case; and we move on to investigate the use of aspects to implement mediators for integrated system design.

4.1 Critique of the *AspectJ* Language Model

As already mentioned, we investigated the mapping of mediators to aspects in an earlier work [33]. In this section we summarize and extend the results presented there.

In a nutshell, mediators can be implemented as aspects in current *AspectJ*-like languages, with one caveat. Most current, major aspect languages, including *AspectJ* and *HyperJ* [35], suffer two shortcomings with respect to the mediator style. First, aspects are essentially global modules, rather than class-like constructs supporting first-class instances under program control. Second, aspects advise entire classes, not object instances. Thus there is, for most practical purposes, effectively *one* instance of each aspect type per system, and it essentially registers with events in *all* instances of each advised class.

By contrast, the mediator style requires that each type of behavioral relationship be represented as a mediator class, with class *instances* representing relationship instances. Moreover, the class instances register with the events of the *instances* to be integrated. Recall, for example, that in our model-view system we have two instances of the *update* mediator, one connecting the model to the first instance of the *view* class, and the other, to the second instance. Each *update* mediator instance registers with the *changed* event of the shared *model* instance, and with the *changed* event of its particular *view* instance.

Mediators cannot be mapped directly to aspects in *AspectJ*-like languages because aspects cannot be instantiated in a general way, nor can they selectively advise instances of other classes. Some aspect languages do support aspect instances or instance-level advising, but they generally have limited join point models (just calls and returns) and limited or no pointcut constructs. Most are in the decades-long tradition of message interception mechanisms. Work-arounds are possible in *AspectJ*, but even the best ones we know incur unnecessary, non-negligible costs in performance or design complexity. The rest of this section analyzes and extends our earlier analysis of this problem [33]. We conclude that straightforward mediator-based design exploiting AOP requires generalizing AOP to the instance-level.

4.2 The Work-Arounds and their Costs

There are two basic work-arounds. In both cases, behavioral relationships types are mapped to aspect modules programmed to simulate first-class aspect instances and instance-level advising. To simulate instances, the aspect provides methods to create, delete, and manipulate instances implemented as records.

The difference is in the simulation of instance-level advising. In the first approach, the aspect advises relevant join points of the classes whose instances are to be integrated. *All* instances invoke the aspect at each such join point. The aspect maintains tables recording the identities of objects to be treated as advised instances. When the aspect is invoked, it looks up the invoker see if it is such an instance. If so, the aspect delegates control to a simulated advice method, or otherwise returns immediately.

This work-around works in the sense that it both modularizes the behavioral relationship code, data, and invokes relations, and relieves the developer of having to work with explicit events. However, the approach is less than ideal for several reasons.

First, it is awkward to have to simulate an object-oriented style in an otherwise object-oriented language. Second, such programs are ironic in the sense that they mean something other than what they literally say, making programs harder to understand: the aspects read as literally advising classes, when, in fact the intent is to advise instances. Third, the approach adds complexity and thus cost and undependability with simulation implementations. Fourth, it adds runtime overhead in two dimensions.

In particular, at each join point, each instance of an advised class has to invoke each advising aspect, if only to have it return upon failing the check for a simulated advised instance. Figure 1 presents data [33] showing that the time to call an empty method increases linearly with the number of aspects that advise calls to the method: any call to the method requires underlying calls to each advising aspect. The overhead of the mediator approach, in contrast, remains constant: a single *if* statement to check to see if any mediators are registered to be invoked. A larger cost is paid only for mediators registered with specific object instances.

There are situations in which the cost of this work-around would be unacceptable. An example would be the case of a mediator implemented as an aspect that has to respond to insertions on just one instance of a widely used, basic *HashTable* class. It would be unreasonable, and is unscalable, for all clients of all instances of *HashTable* to have to pay a price for that one, isolated client.

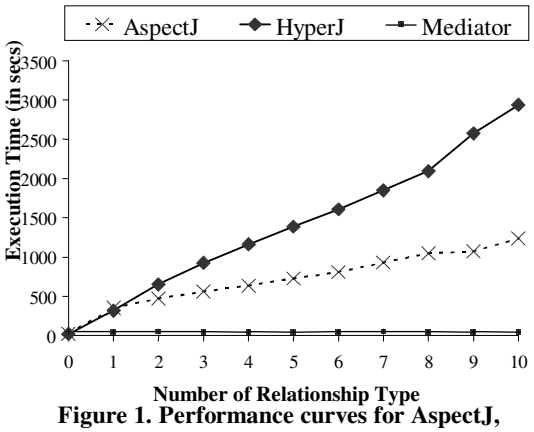

Figure 1. Performance curves for AspectJ, HyperJ and Mediator based Design.

The second work-around uses *AspectJ introduction* to extend the classes to be integrated with explicit event interfaces and code. The aspect also advises the join point at which the event is to be announced. The advice announces the event. The objects registered are not themselves aspects but ordinary mediators. The effect is to implement a traditional mediator design, but with explicit events modularized with the mediators that need them.

This work-around works, too, achieving integration without loss of modularity. It also avoids the performance overhead of the first work-around by using the same event mechanisms that mediators use. Finally, it relieves the component developer of having to anticipate the events that mediators might need. In that sense, it arguably improves on the original mediator style.

Yet the approach has some problems. First, it doesn't really implement mediators as aspects at all, but only modularizes the explicit events that mediators need. It misses the point: we want to use join points *rather than* explicit events to invoke mediators. Having join points invoke advice that announces events that invoke mediators is at best a complex, relatively costly approach, using redundant mechanisms (events, join points). Second, if several mediators need to respond to the same event, each introduces its own event code and interface—bloating the code—rather than using the same event or join point.

5. A NEW ASPECT LANGUAGE DESIGN

The aspect-inspired critique of mediators, the idea of mediators as aspects, and the subsequent mediator-based critique of *AspectJ*-like languages, leads to our main contributions. We generalize *AspectJ*-like languages to the instance-level, and show that doing so enables (among other things) a clean and direct mapping of mediators to aspects. We exhibit the ideas in a novel *AspectJ*-like language called *Eos*, which is based on C# [28].

From the programmer's view the key difference between *AspectJ* and *Eos* (ignoring host language differences) is in added support for first-class aspect instances and instance-level advising. These aspects can be instantiated and instances can advise objects selectively. The underlying mechanism weaves implicit invocation structures into advised code at join points identified by pointcut expressions, enabling selective, runtime registration of individual aspect instances. The use of an event mechanism is abstracted from the programming model and is entirely invisible to the language user.

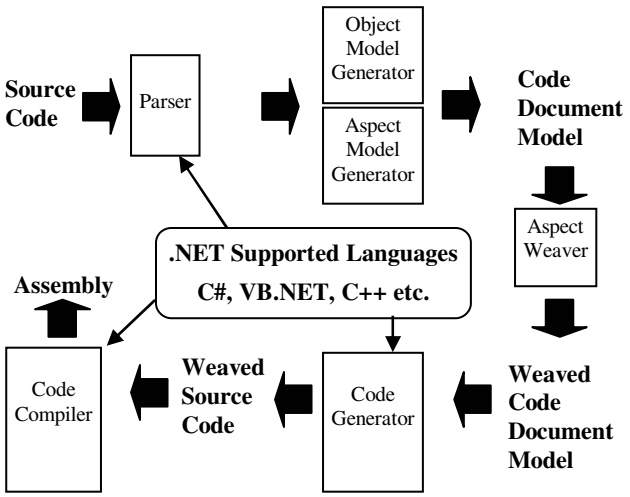

Figure 2. Architechture and Working of *Eos*

5.1 Compiler Architecture

We implemented a prototype *Eos* compiler using a custom version of the Code Document Object Model (CodeDOM) [30], part of the base class library of Microsoft's .NET framework. CodeDOM provides a language independent object model for representing and rendering source code in supported languages. A CodeDOM model is essentially an abstract syntax tree (AST). CodeDOM allows programs to be dynamically created, compiled, and executed at runtime.

Figure 2 shows the components of our prototype *Eos* compiler. A tokenizer (not shown) extracts tokens from the source code and passes them to the parser. The parser generates a CodeDOM AST for the source code and aspect code. The aspect weaver takes the AST and weaves aspects into it. Type-level aspects are woven statically. Instance-level aspects result in the weaving of event stubs for runtime registration of advice. If the source code generation is specified, the code generator generates woven source code; otherwise the AST is compiled directly into an *assembly*—an executable program in the.NET framework.

An element not entirely unique to our approach is that weaving is done on a language independent structure. *Eos* and AspectC# [23] use CodeDOM. CLAW [26] uses Microsoft Intermediate Language (MSIL), which is analogous to byte code in Java [18]. This decision enables weaving of aspects written in one language (e.g. C#) into code written in another (e.g. Visual Basic). We have not yet explored this possibility in any detail.

We chose to weave source code, instead of using MSIL, to ease prototype development. CodeDOM provides a rich applications programming interface (API) to represent code as an object graph and for compiling such a graph into a supported language. Implementing the prototype using these libraries was relatively easy. Some constructs of C#, including namespaces, aliasing, operator overloading, and others—are not supported by the current CodeDOM, limiting the *Eos* prototype to a useful but not exhaustive subset of *C#*.

5.2 *Eos*: Syntax and Semantics

Like AspectJ for Java, *Eos* adds to C# *join points*, *pointcuts*, *advice*, *introductions* and *aspects*. The *Eos* syntax extends the C# syntax as defined in the ECMA *C#* language specification [28]. *Eos* adds the following key words:

advice after any args around aspect
before call execution fget fset instancelevel
pointcut returning throwing

Most of these keywords are similar to their *AspectJ* counterparts. Keywords *fget* and *fset* are equivalent to *get* and *set* of AspectJ to avoid conflict with *C# get* and *set*. Keyword *any* is equivalent to *AspectJ's* "*" to avoid conflict with the C# pointer "*" operator.

In C#, a reference type is a class type, interface type, array type, or delegate type [28]. *Eos* extends this set to include aspect types. An aspect type defines a data structure containing data members (constants and fields), function members (methods, properties, events, indexers, operators, instance constructors, destructors, and static constructors), crosscutting members (pointcuts, advices and introductions) and nested types. Aspect types support inheritance, whereby derived aspects can extend abstract aspects or classes. A class may not extend an aspect. Instance-level aspects in *Eos* support first class objects. They can be instantiated, passed as arguments, returned as results, etc. Instances are created using C# *object-creation-expressions* [28]. Type-level aspects cannot be instantiated.

An aspect-modifier is either of the permissible class-modifiers or the keyword *instancelevel,* which specifies instance-level aspect weaving. As in C#, it is a compile-time error for a modifier to appear multiple times. The *instancelevel* modifier can be applied to the aspects and advices as illustrated in the following code:

```
1 public aspect A {  /* Some Members */ } /// Type-level aspect in Eos
2 public instancelevel aspect B { // Instance-level aspect in Eos
3    pointcut callfoo():call(public void any.foo());
4  }
5 public aspect C { // Aspect in Eos containing both type and instance-level advices
6    pointcut fval(): fget(public int any.val);
7    instancelevel after(): execution (public void any.bar());
8    after():fval() { // Do something }
9  }
```

Here, aspect A has no weaving modifier so its advice is woven at the class level: aspect A advice, woven into a class P, affects all instances of P. The *instancelevel* modifier delays advice weaving until runtime. Aspect B above is an example. At compile time, the weaver attaches event stubs at the join points matched by the pointcut declaration, to enable instance-level, dynamic weaving. One might want to weave some advice at the class level and other advice at the instance level. *Eos* allows this kind of mixing. For example, aspect C has no weaving modifier so its advice is woven at type level except for that designated as *instancelevel*.

In *Eos,* like AspectJ, pointcuts are used to identify sets of join points. For example, the pointcut *call(public any.bar())* identifies any call to the method bar defined by any class. *Eos* also provides operators and (&&), or (||) and not (!) to compose pointcuts. The pointcut *call(public void any.bar()) || call(public void any.foo())* thus identifies any call to either the bar or the foo methods defined by any classes. Similarly, pointcut *call(public void any.bar()) && call(public void any.foo())* identifies any call to the bar and the foo methods, which is an empty set.

Our *Eos* prototype design provides a rudimentary mechanism for stating which objects are subject to instance-level adving. An aspect declared as *instancelevel* or containing *instancelevel* advice provides implicit methods *addObject* and *removeObject* for specifying instances to be advised. These methods basically implement (un)registration with the underlying event structures.

Advice code can access reflective information at join points using the implicit argument *thisJoinPoint*. Depending on the join point, the methods *getThis*, *getTarget*, *getReturnValue*, *getArgs* returns this object, the target object, the return value (only for method call and execution join points), and method arguments (for method call and execution join points) respectively.

To illustrate these ideas we describe a simple *Eos* system, no longer distinguishing models and views. Suppose you are asked to build a system of *n* model instances, m1, m2, …, each an instance of a *Model* class.

```
1   public class Model  {
2    bool value;
3    public Model() { value = false; }
4    public void Set() { value = true; }
5    public bool Get (){ return value; }
6    public void Clear() {value= false; }
7   }
```

There are several types of relationships on models, and a given model instance can be in zero or more relationship instances. One relationship is *Consistency*. It requires that if a client *Sets* or *Clears* either model, the other must be *Set* or *Cleared,* too. The following implements *Consistency* as an instance-level aspect.

```
1  public instancelevel aspect Consistency  {
2   Model m1, m2;
3   bool busy;
4   public Consistency(Model m1, Model m2)   {
5              addObject(m1);  addObject(m2);
6              this.m1 = m1;    this.m2 = m2;
7              busy = false;
8   }
9   after():execution(public void Model.Set ())   {
10     if(!busy) {
11             busy = true;
12             Model m = (Model) thisJoinPoint.getTarget();
13             if(m == m1)m2.Set(); else m1.Set();
14             busy = false;
15             }
16   }
17   after():execution(public void Model.Clear ())   {
18     if(!busy) {
19             busy = true;
20             Model m = (Model) thisJoinPoint.getTarget();
21             if(m == m1)m2.Set(); else m1.Set();
22             busy = false;
23             }
24   }
25  }
```

Line 1 declares the aspect as *instancelevel*. The implicit method *addObject* is used on line 5 to register advice with objects *m1* and *m2*. The aspect declares two advices (lines 9-16 and 17-24). The first executes after the method *Set* in class *Model*; the second, after *Clear*. Lines 12 and 20 use reflective information, via *thisJoinPoint,* to find out which model is being set or cleared.

Now consider a relationship on models, *Trigger(m1, m2),* where *m1* and *m2* play different roles*: when* m1 or m2 *is* Set*,* m2 *or* m1, *respectively, must be* Set*; and—here's the asymmetry--if* m1 *is* Cleared*,* m2 *must be cleared*. Thus *m1.Clear* acts as a trigger with *m2* as its target. But *m2.Clear* has no effect on *m1*.

To implement such a relationship, we need to attach advice for different roles to different objects. *Eos* provides the *role* construct for this purpose. The following code implements the *Trigger* relationship using *roles*:

```
1  public instancelevel aspect Trigger {
2  Model m1, m2;
3  bool busy;
4  public Trigger(Model m1, Model m2){
5          this.m1 = m1; this.m2 = m2;
6          AddRoleTrigger(m1); AddRoleTarget(m2);
7          }
8   role Trigger {
9     after():execution(public void Model.Set()) {
10           if(!busy){ busy = true;  m2.Set();  busy = false; }
11           }
12    after():execution(public void Model.Clear()) {
13          if(!busy){  busy = true; m2.Clear(); busy = false; }
14           }
15    }
16  role Target {
17     after():execution(public void Model.Set()) {
18          if(!busy){ busy = true; m2.Set(); busy = false; }
19          }
20     }
21  }
```

This instance-level aspect defines two roles, *Trigger* and *Target*. An object is put in a role *X* using the generated implicit method, AddRole*X*, which essentially registers the role advice with the given instance. Here, whenever *Set* is called on *m1,* the advice in the role *Trigger* will execute, whereas when these methods are called on instance *m2,* advice in the role *Target* will execute.

5.3 The *Eos* Aspect Weaver

To explain how the *Eos* weaver works, we start by reviewing the *AspectJ* weaver (version 1.0.6 [8]). Suppose a *Model* class in *AspectJ* is advised by *Consistency* and *Trigger* aspects. The woven *Model* code would appear as follows:

```
1  /*   Generated by AspectJ version 1.0.6 */
2   public class Model {
3    static org.aspectj.runtime.reflect.Factory ajc$JPF;
4    private static org.aspectj.lang.JoinPoint.StaticPart Set$ajcjp1;
5    private static org.aspectj.lang.JoinPoint.StaticPart Clear$ajcjp2;
6    boolean value;
7    public Model() { super();  this.value = false;  }
8    public void Set() {
9      final org.aspectj.lang.JoinPoint thisJoinPoint =
10       org.aspectj.runtime.reflect.Factory.makeJP(
11       Model.Set$ajcjp1, null, this, new java.lang.Object[] { });
12     try { this.value = true; } finally {
13      Consistency.aspectInstance.after0$ajc(thisJoinPoint);
14      Trigger.aspectInstance.after0$ajc(thisJoinPoint);
15     }
16  }
17  public void Clear() {
...        // Similar to Set
25    }
26   public boolean Get() { return this.value;  }
27   static {
28   Model.ajc$JPF = new org.aspectj.runtime.reflect.Factory("Bit.java", Bit.class);
29   Model.Set$ajcjp1 = Model.ajc$JPF.makeSJP("method-call",
30       Model.ajc$JPF.makeMethodSig("1-Set-Model----void-"), null);
31    Model.Clear$ajcjp2 = Model.ajc$JPF.makeSJP("method-call",
32       Model.ajc$JPF.makeMethodSig("1-Clear-Model----void-"), null);
33   }
34 }
```

AspectJ achieves implicit invocation at the type-level by inserting calls to the *after* advice of *Consistency* and *Trigger* into the *Model* Code. The points at which invocation code is inserted are determined by the pointcut expressions.

Eos, by contrast, statically determines what join points *might* be advised by instance-level constructs and instruments them not with calls to the aspect advice but with events supporting dynamic registration and event announcement. The *Eos* woven code for *Model* follows:

```
1  /*     Generated by Eos version 0.1 */
2   public class Model {
3   bool value;
4   public Model() {    value = false;    }
5   public void Set() {
6        eos.Joinpoint thisJoinPoint =
7            new eos.Joinpoint(null,this, null, new System.Object[] { });
8        try {  value = true;  } finally {
9            if(ADP_EOS_After_EXECUTION_Set!=null)
10               ADP_EOS_After_EXECUTION_Set(thisJoinPoint);
11       }
12   }
13  public bool Get() {   return value;   }
14  public void Clear() {
...       // Similar to Set
21   }
22  public event eos.ADP ADP_EOS_After_EXECUTION_Set;
23  public event eos.ADP ADP_EOS_After_EXECUTION_Clear;
24  }
```

Here, *Eos* determines that two join points might be advised: after *Set* and *Clear*. Two events, ADP_EOS_After_EXECUTION_Set and ADP_EOS_After_EXECUTION_Clear, are introduced, and are announced after execution of the Set and Clear method bodies. The implicit *addObject* and *removeObject* methods of the *Consistency* and *Trigger* aspects (un)register advice with these events to achieve instance-level weaving. The generated code for the *Consistency* aspect follows:

```
1  /*     Generated by Eos version 0.1 */
2  public class Consistency  {
3    Model m1, m2;
4    bool busy;
5    public Consistency(Model m1, Model m2)   {
6              addObject(m1);  addObject(m2);
7              this.m1 = m1;    this.m2 = m2;
8              busy = false;
9   }
10  public void EOS_Advice_After0(eos.Joinpoint thisJoinPoint)   {
11      if(!busy) {
12             busy = true;
13             Model m = (Model) thisJoinPoint.getTarget();
14             if(m == m1)m2.Set(); else m1.Set();
15             busy = false;
16             }
17   }
18  public void EOS_Advice_After1(eos.Joinpoint thisJoinPoint)   {
...             // Similar to the first advice
25   }
26  public void addObject(object obj) {
27     if(obj == null)return ;
28     if(obj is Model) {
29          Model casted_obj = ((Model)(obj));
30          casted_obj.ADPart_EOS_After_EXECUTION_Set +=
31                new eos.ADP ( EOS_Advice_After0);
32          casted_obj.ADP_EOS_After_EXECUTION_Clear +=
33                new eos.ADP ( EOS_Advice_After1);
34          }
35     }
36  public void removeObject(object obj) {
37     if(obj == null)return ;
38     if(obj is Model){
39          Model casted_obj = ((Model)(obj));
40          casted_obj.ADP_EOS_After_EXECUTION_Set -=
41                new eos.ADP( EOS_Advice_After0);
42          casted_obj.ADP_EOS_After_EXECUTION_Clear -=
43                new eos.ADP( EOS_Advice_After1);
44          }
45     }
46  }
```

6. EVALUATION OF CONTRIBUTIONS

In this section we evaluate our work, answering the following questions. Does the compiler work? Does this research advance our ability to design integrated systems using mediators, with join points and pointcuts in place of explicit events, but without the performance and complexity penalties imposed by available work-arounds in *AspectJ*-like languages? Does the work advance our understanding of the design of *AspectJ*-like languages?

In a nutshell, the compiler works. *Eos* supports first-class aspect instances and instance-level weaving, enabling a clean mapping of mediators to instance-advising aspect instances. There is no need for simulation of instance-level constructs or explicit event code. The performance of instance-level aspects is comparable to that of the mediator style. Our work does therefore advance the mediator approach. This work also shows that there is at least one good reason to generalize *AspectJ*-like languages to the instance level, and it provides a clean proof-of-concept language design, implementation, and demonstration.

6.1 Runtime Performance

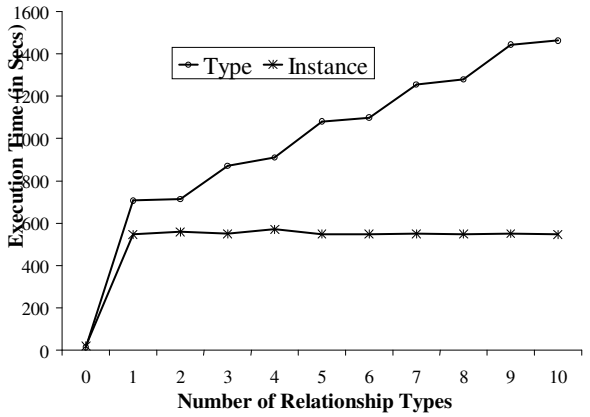

Figure 3. Performance curves for type-level and instance-level constructs of *Eos*.

We measured the performance of instance-level aspects using the same benchmarks as used above to evaluate *AspectJ*, *HyperJ* and mediator-based designs. The comparison is complicated by the differences in host languages (Java, *C#*). We substitute type-level *Eos* aspects for *AspectJ* aspects for this comparison. Figure 3 shows that *Eos* type-level aspects replicate the degrading performance of *AspectJ* aspects, while *Eos* instance-level aspects indeed exhibit the constant overhead of mediator-based designs.

6.2 Implementing Complex Mediator Designs

To test the hypothesis that *Eos* supports the design of realistic systems using aspect instances as mediators, we implemented, in *Eos,* two key mediator structures used in the design of *Prism,* an integrated environment for radiation treatment planning—itself a major test of the mediator approach [19][34]. The first mediator maintains a bijection between two sets; the second, a cross-product between two sets. These fragments played important roles in Prism, and are representative of the kinds of mediators that arise when the approach is used to design real systems.

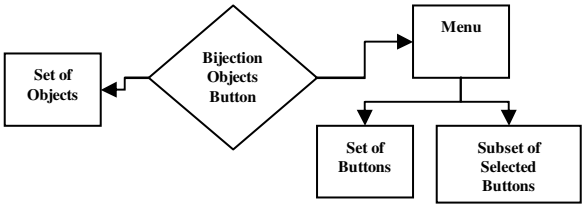

Figure 4. A mediator-based model-view system [34]

A common requirement is to maintain consistency between sets of GUI objects (buttons, menu items, shapes) and sets of model objects (in Prism, models of internal organs of cancer patients). In Prism, one or more menu items can be selected to open panels for editing the designated model objects. We analyze this system as a set of model objects, a set of menu items, a selected subset of menu items, a relationship that keeps the set of model objects and the set of menu items in one-to-one correspondence, and a relationship that keeps the subset of selected menu items consistent with a set of individual model editing panels.

Figure 3 presents a schematic showing the first relationship. Sets and menus are implemented as instances of *Set* and *Menu* types with operations to insert, delete and iterate over elements. In the original mediator implementation, these classes had explicit events. *Eos* eliminates this requirement. The key parts of the *Eos Set* and *Bijection* mediator implementation are as follows.

```
1  public class Set : System.Collections.CollectionBase {
2     public bool Insert(Element element){ … }
3     public bool Remove(Element element){ … }
4     public Element Retrieve(string Name){ … }
5     }
6  public instancelevel aspect Bijection {
7       bool busy; Set A; Set B;
8       public void Bijection(Set A, Set B){
9             addObject(A); addObject(B);
10              this.A = A; this.B = B;
11          }
12    after():execution(public bool Set.Insert()){
13             if(!busy && (bool) thisJoinPoint.getReturnValue()) {
14               busy = true;
15                …          /// Bijection Logic: Ensures bijection between the sets.
16               busy = false;
17               }
18          }
19    after():execution(public bool Set.Remove()){
20          if(!busy && (bool) thisJoinPoint.getReturnValue()) {
21               busy = true;
22               …  /// Bijection Logic: Ensures bijection between the sets.
23              busy = false;
24               }
25       }
26    }
```

The *Set* class exposes no explicit events. The *Bijection* mediator, implemented as an instance-level aspect, provides a method *Relate* to relate two instances of the *Set* type. It uses the implicit *addObject* methods to "register" the advice with these instances. The aspect uses pointcuts *execution(public bool Set.Insert())* and *execution(public bool Set.Remove())* to identify the *Insert* and *Remove* events. The join points are instrumented with the required events. When the events occur, the advice is invoked to keep the sets consistent (and to update a mediator-maintained table recording associations between model and view objects).

The cross product mediator is similar to the bijection mediator but it maintains a cross product of two sets. Prism uses such a mediator to ensure that each model in a set of models is depicted in each view in a set of views. As views or objects are added and deleted, the cross product is maintained in a consistent state. Moreover, Prism keeps corresponding pairs of objects consistent by creating a "sub-mediator" for each pair. The following code is an *Eos* implementation, with the *Set* type as defined above.

```
1  public instancelevel aspect CProduct : System.Collections.CollectionBase {
2    bool busy; Set A; Set B;
3    public void CProduct(Set A, Set B){
4              addObject(A); addObject(B);
5              this.A = A; this.B = B;
6              }
7    after():execution(public bool Set.Insert()){
8              if(!busy&&thisJoinPoint.getReturnValue()){
9                  ... // Ensure Cross product Logic
10                 OrderedPair op = new OrderedPair(.. , .. );  //Dispatch the sub
11                             // mediator with corresponding elements in ordered pair.
12             }
13   after():execution(public bool Set.Remove()){
14             if(!busy &&thisJoinPoint.getReturnValue()){
15                 ... //Ensure Cross Product Logic
16           // Remove the elements from the sub mediator ordered pair and then
17             // remove the submediator itself.
18             }
19   }
20   public void Add(OrderedPair op){
21             /* Add the ordered pair to collection*/
22   }
23   public bool Remove(Element A, Element B){
24              /* Remove the ordered pair (A,B) from the collection */
25   }
26 }
27 public instancelevel aspect OrderedPair {
28   public String Name; bool busy; Element A; Element B;
29   pointcut fieldSet():fset(System.String Element.Name);
30   public void Order(Element A, Element B){
31             addObject(A); addObject(B); this.A = A; this.B = B;
32             }
33   public void RemoveOrder(){
34             removeObject(A); removeObject(B);
35   }
36   after():fieldSet(){
37             if(!busy){
38             /// Ensure consistency of the elements.
39             }
40   }
```

Networks of objects integrated by mediators are also supported. Consider a network in which two *Sets* A and B are integrated by a *Bijection*, with B and C are integrated by a *CrossProduct*. This can now be accomplished by the following *Eos* program. We are satisfied that *Eos* provides essentially full support for mediator-based design.

```
1  public class TestHarness {
2    public static void Main (string[] argument) {
3              Set A = new Set();  Set B = new Set(); Set C = new Set();
4              Bijection bj = new Bijection(A, B); // Integrate A and B
5              CrossProduct cp = new CrossProduct(B, C); // Integrate B and C
6              }
7                                                                          }
```

6.3 An Advance in Aspect Language Design

We have identified a need to generalize *AspectJ*-like languages to the instance level. The use of aspects in practice is still so limited that few confirmatory data are available. Griswold et al. reported (personal communication) that in AspectBrowser [7], of 54 aspects appearing in 15,000 lines of *AspectJ* code, 11 (20%) employed the simulation work-around strategy described above.

The work-arounds do work, but at a cost of conceptual integrity, code complexity, and runtime performance. Moreover, there are cases where the simulation approach is not acceptable: those in which a small number of instances of widely used classes need to be advised. Imposing simulation costs on all instances is less than ideal, at best. One can invoke a magic compiler to take care of the problem, but it's not clear how it would actually work.

We have shown that work-arounds are not necessary, and that a solution—in first-class aspect instances and instance-level advising—can be integrated into *AspectJ*-like languages, with a principled implementation based on an underlying implicit invocation system. We know of no other *Eos*-like languages.

7. RELATED WORK

Instance-level advising is not a new idea. Many examples take a wrapper-based approach: *AspectS* [17], Composition Filters [10], the Aspect Moderator Framework [11], the Object Infrastructure Framework [14], and *EAOP* [13], in which messages sent to and from components are intercepted for processing by aspect wrapper objects. The idea has appeared in many forms over the decades: from Common Lisp, to tool integration frameworks. It has been picked up and given an AOSD interpretation by efforts such as Sina/st[24][5] and ility-insertion [14].

Many such languages are called *aspect-oriented.* However, they are not in the class we call *AspectJ*-like. In particular, they generally have very limited join point models (mostly message interception) and limited or non-existent pointcut languages. Other *AspectJ*-like languages, such as *AspectC++* [3] and *AspectR* [4], on the other hand, do not support first-class aspect instances and instance-level advising. Languages such as *HyperJ* [35] and *DJ* [27], in which the pointcut concept does not apply, also lack the instance-level capabilities of *Eos*.

The idea of having one component modify the behavior of another based on its role in a system is not new, either. A recent incarnation appears in work on roles and role models [6][16][25]. Kendall [20][21] saw that aspect instances might be used to modify object instances based on their roles, and demonstrated the idea using a constrained forms of aspect instances available in various versions of *AspectJ:* the current *per this* and *per target* constructs, and the version 0.6 *addObject* method, which allowed one to attach an aspect to an instance as opposed to type.

For our purposes, these constructs are or were inadequate. First, *per this* and *per target* aspect instances are not under program control but are associated automatically with all instances of advised types; and each instance is associated with just one other object. Mediators, in general, have to advise several objects, and represent relationships that can be imposed and retracted.

AspectJ supported *addObject* and *removeObject* through version 0.5. The AspectJ mailing list [9] records discussions on the need for *addObject*, with examples presented by De Luca, van Gurp, and others. Thus the idea of instance-level advising in an *AspectJ*-like language is not new. However, *AspectJ*, to our knowledge, never supported both instance-level advising and first-class aspect instances. This is the combination needed for a complete generalization to the instance level. Our mediator-based critique shows the need, and *Eos* demonstrates a novel solution.

Coordination contracts [1] are related to both aspect oriented languages and mediators. A coordination contract represents a behavioral relationship, and can intercept method calls and introduce new behavior at the instance level. However, the join point model is, again, basically limited to the method calls.

Nor is the idea of an aspect version of C# new. AspectC# [23] supports class advising. Cross Language Aspect Weaving (CLAW) [26] borrows from *AspectJ* and supports dynamic class advising. AOP# [2], developed at Siemens, aims to support class level advising without language extensions. Other approaches are Aspect.Net [29] and Aspect-Oriented Infrastructure for a Typed, Stack-based Intermediate Assembly Language [12]. We chose C# mostly for ease of prototype language development.

Finally, we note that events (implicit invocation, publish-subscribe) are not new. We make no advances here, but do show that events can provide a useful runtime mechanism, abstracted by higher-level join point and pointcut language constructs.

8. CONCLUSION

Integrated systems provide value but are often costly to develop and maintain because common design methods map integration requirements to non-modular designs. The mediator approach largely modularizes behavioral relationships, but with scattered event declaration, registration, and announcement code. *AspectJ*-like languages promise to reduce this cost by mapping mediators to aspects and using join points in place of events. The problem is that these languages do not support two key features needed for a clean mapping of mediators, first-class aspect instances and instance-level advising. We generalized *AspectJ*-like languages to include these features, presented *Eos*, and showed that it does enable mediator-based design without costly work-arounds.

A disadvantage of *Eos* is that, although its join point model is rich relative to many languages said to be aspect-oriented, it is nevertheless limited. A benefit of explicit events is that they can be declared at will and can be given arbitrary semantics. For example, a mediator might have to respond if one branch of an *if* statement is taken but not the other (e.g., representing successful insertion of an element into a collection). In Prism, such events were routine. *AspectJ*-like languages do not expose such events as join points. More generally, the designer of an aspect language effectively anticipates what classes of events might be relevant. Here the aspect critique of mediators returns to haunt the aspect language designer: it's hard to anticipate.

Our next task is therefore to generalize *AspectJ*-like languages to expose a far wider set of execution phenomena as join points. In the extreme, every significant event in the operational semantics of the language becomes visible as a join point. A challenge will be to find reasonable ways to name them using pointcuts.

9. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant ITR-0086003. We thank Bill Griswold, for data on *AspectBrowser*; Gregor Kiczales, for discussions on the aspect critique of the mediator style; and Michael Jackson, for the idea that aspect languages might usefully expose essentially all semantically meaningful execution phenomena as join points.

10. REFERENCES

[1] Andrade, L., and Fiadeiro, J.,"Coordination Technologies for Managing Information System Evolution", Proceedings of CAiSE'01, K.Dittrich, A.Geppert and M.Norrie (eds), LNCS 2068, Springer-Verlag 2001, 374-387.

[2] AOP#: under development at Siemens Corporation. Egon.Wuchner@mchp.siemens.de.

[3] AspectC++, http://www.aspectc.org.

[4] AspectR: "Simple Aspect Oriented Programming in Ruby," http://aspectr.sourceforge.net/

[5] Aksit, M., Wakita, K., Bosch, J., Bergmans, L., Yonezawa, A., "Abstracting Object Interactions Using Composition Filters," Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming, 1993.

[6] Andersen, E. (Egil), *Conceptual Modelling of Objects: A Role Modelling Approach*, PhD Thesis, University of Oslo, 1997.

[7] AspectBrowser: http://www-cse.ucsd.edu/users/wgg/swevolution.html

[8] AspectJ: www.eclipse.org/AspectJ

[9] AspectJ mailing list archives: http://www.eclipse.org/AspectJ

[10] Bergmans, L., "The Composition Filters Object Model," Dept. of Computer Science, University of Twente, 1994.

[11] Constantinides, C., A., and Elrad, T., "Composing Concerns with a Framework Approach," Proc. 2nd Int'l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS-2002), Vol. 2, July, 2002, Vienna pp. 133-140.

[12] Dechow, D., Ph.D Dissertation Proposal, http://cs.oregonstate.edu/~dechow

[13] Douence, R., and Südholt, M., "A model and a tool for Event-based Aspect-Oriented Programming (EAOP)", TR 02/11/INFO, École des Mines de Nantes, french version accepted at LMO'03, 2nd edition, Dec. 2002.

[14] Filman, R. E., Barrett, S., Lee, D. D. and Linden, T., "Inserting Ilities by Controlling Communications " , Communications of ACM, vol. 45, number 1, Jan, 2002, pp. 116-122.

[15] Garlan, D., and Notkin, D., "Formalizing Design Spaces: Implicit Invocation Mechanisms". *VDM '91: Formal Software Development Methods*, pp. 31--44 (October 1991).

[16] Gottlob, G., Schrefl, M., and Rock, B., "Extending Object Oriented Systems with Roles," ACM Trans on Info. Sys., Vol. 14, No. 3, July, 1996, pp. 268 - 296.

[17] Hirschfeld, R., "AspectS -- Aspects in Squeak", ECOOP'2002 Workshop on Generative Programming, Jun 2002.

[18] Java: www.java.sun.com

[19] Kalet, I.J., J.P. Jacky, M.M Austin-Seymour, S.M. Hummel, K.J. Sullivan and J.M. Unger, ``Prism: a New Approach to

Radiotherapy Planning Software," *International Journal of Radiation Oncology, Biology and Physics, 36*, 2, 1996, pp. 451--461.

[20] Kendall, E. A., "Aspect Oriented Programming for Role Models," International Workshop on Aspect Oriented Programming, European Conference on Object Oriented Programming (ECOOP'99), Lisbon, June, 1999.

[21] Kendall, E. A., "Aspect-oriented Programming in AspectJ," Evolve 2000, Sydney, March, 2000.

[22] Kiczales, G.., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. and Irwin, J., "Aspect-oriented programming," in Proceedings of the European Conference on Object-Oriented Programming (ECOOP), Springer-Verlang, Lecture Notes on Computer Science 1241, June 1997.

[23] Kim, H., "AspectC#: An AOSD implementation for C#," Technical Report TCD-CS-2002-55, Department of Computer Science, Trinity College, Dublin, 2002.

[24] Koopmans, P.S., "Sina/St: User's Guide and Reference Manual," TRESE project, University of Twente, 1996.

[25] Kristensen, B. B., "Object-oriented Modelling with Roles," OOIS'95, Proceedings of the 2nd International Conference on Object-oriented Information Systems, Dublin, Ireland, 1996.

[26] Lam, J., "CLAW" URL: www.iunknown.com.

[27] Marshall, J., Orleans, D., and Lieberherr, K., "DJ: Dynamic Structure-Shy Traversal in Pure JAVA," Technical Report, Northeastern University, May 1999.

[28] Microsoft. C# Specification Homepage. http://msdn.microsoft.com/net/ecma/.

[29] Microsoft Aspect.Net project description. http://research.microsoft.com/programs/europe/rotor/Projects.asp.

[30] Microsoft .Net Framework Developers Guide available at http://msdn.microsoft.com

[31] Sullivan, K., "Mediators: Easing the Design and Evolution of Integrated Systems", Ph.D. dissertation, University of Washington, 1994.

[32] Sullivan, K. and Notkin, D., "Reconciling environment integration and software evolution," ACM Transactions on Software Engineering and Methodology 1, 3, July 1992, pp. 229–268 (short form: Proceedings of the 4th SIGSOFT Symposium on Software Development Environments, 1990, pp. 22–33).

[33] Sullivan, K., Gu, L., Cai, Y., "Non-modularity in Aspect-Oriented Languages: Integration as a Crosscutting Concern for AspectJ," Proceedings of Aspect-Oriented Software Design, 2002

[34] Sullivan, K., Kalet, I., Notkin, D., " Evaluating the mediator method: Prism as a case study," *IEEE Transactions on Software Engineering, Vol. 22*, No. 8, August 1996.

[35] Tarr, P. and Ossher, H., "Hyper/J™ User and Installation Manual", IBM Corporation