

Need for Instance Level Aspect Language with Rich Pointcut Language

Hridesh Rajan
hr2j@cs.virginia.edu

Kevin Sullivan
sullivan@cs.virginia.edu

Department of Computer Science
School of Engineering, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740

1. Introduction

A potentially important part of the design space for aspect-oriented languages remains largely unexplored: namely that space of languages that combine *expressive pointcut languages* and *instance level aspect weaving*. We define instance level aspect weaving as the ability to differentiate between two instances of a class and to weave them differently if needed. Note that instance level aspect weaving and dynamic/run-time aspect weaving are not the same, former allowing the developer to differentiate between instances whereas the later gives him/her the ability to attach and detach aspects at run-time. In summary, instance level aspect weaving allows you to say, “weave this aspect to only these instances of that class”.

Languages such as AspectJ [4] have expressive pointcut sublanguages but do not support aspect weaving at instance level. Rather, aspects modify programs at the type level, affecting all instances of that type. Approaches such as the Aspect Moderator Framework [7] and Object Infrastructure Framework [13], by contrast, can weave aspects at the instance level but lack expressive pointcut languages. We believe there is significant potential value in models having both elements. We are thus investigating the feasibility and the utility of AO languages having both elements.

The rest of the paper is organized as follows. Section 2 presents an analysis of the space of existing languages. Section 3 outlines our case for the need to explore the empty quadrant. Section 4 describes an early prototype language design and implementation, *ilaC#*, an instance level aspect language for C#. Section 5 presents related work. Section 6 concludes.

2. Characterization of AOP Languages

Fig. 1 presents taxonomy of aspect-oriented languages in two dimensions: level of weaving and richness of pointcut language. The space of interest is shaded gray.

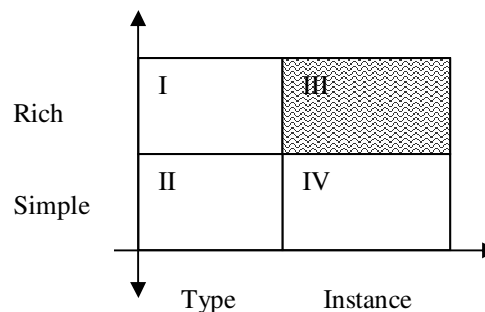


Fig 1: Characterization of Aspect Oriented Languages

The horizontal axis distinguishes languages based on the level of weaving: are aspects necessarily woven into types, and thus into all instances of affected types, or can aspects affect object instances selectively? The Y-axis, by contrast, distinguishes aspects languages by the richness of their pointcut sublanguages. The language design space is thus partitioned into four sub-spaces.

Existing languages fall more or less clearly in these spaces. In AspectJ [4], for example, aspects modify program behaviors at the type level (with the exception of the legacy, effectively deprecated, and quite limited *per this* and *per target* capabilities). However, AspectJ provides an especially rich pointcut sublanguage. AspectJ therefore lies in quadrant I.

HyperJ [27], as implemented, lies in the sub-space II, but its join point model is significantly more expressive so as a concept it should be placed in sub-space I. It too weaves at the type level only, and has a significantly less expressive language for selecting sets of join points to be advised by an aspect. It too weaves at the type level only, and has a significantly less expressive language for selecting sets of join points to be advised by an aspect. Its fundamental advantage over AspectJ is in its uniform treatment of modules as potentially crosscutting, in contrast to the distinction that AspectJ's draws between base object-oriented and aspect code. This distinction, however, is orthogonal to the issue we address, so we do not discuss it any further.

Instance level aspect modification of program behavior is possible in the Aspect Moderator Framework [7] and the Object Infrastructure Framework [13]. However, they lack expressive pointcut language constructs, complicating the specification of crosscutting sets of join points. Moreover, their join point models appear to be limited to methods pre- and post-activation. Based on these characteristics the frameworks lie in sub-space IV.

Event-based Aspect Oriented Programming (EAOP) [9] similarly falls in quadrant IV. Its aspect composition mechanism allows aspects to modify aspects, and it supports the creation, removal, and reorganization of aspects at run-time as well as instance level aspects. However, at this time, EAOP appears to lack a pointcut sublanguage. Thus, EAOP falls in quadrant IV.

The composition filters model—at present apparently not available in implemented form—is a class-based model [6] that is said to support instance level aspects. However, it does not offer a rich pointcut language. Based on our characterization it lies in sub-space IV.

A survey of the languages we have discussed, and others, reveals an apparent void in quadrant III. There are few if any languages currently offering both instance level weaving and rich pointcut languages. Our position is that languages in this class have the potential to provide value beyond that of current aspect languages.

Of course, there is a reason that such languages have not emerged: The combination of the dimensions raises significant and not fully resolved technical problems in language design and implementation. The contribution of this position paper is to have identified the issue. Our plan is to address it. The rest of this paper provides both a motivation for investing effort in this area and a brief report on progress to date, including an early prototype language implementation based on C#.

3. Need for Instance Level Aspects

Supporting instance-level “aspects” that extend program behavior using method-call-intercepting wrappers is reasonably well understood. The wrapper approach has been investigated under various names and guises for at least two decades. However, deeper issues remain unresolved concerning language support for instance-level aspects in languages such as AspectJ and HyperJ. To investigate the nature and extent of the problem in practice, we undertook several preliminary studies.

First, we showed that although it seemed reasonable to use aspects in place of implicitly invoked aspect-like mediators to effect component integration [24], the limits of AspectJ precluded the use of the powerful features of the language to achieve a fully satisfactory modular solution [25].

The mediator approach models behavioral relationships, or potential collaborations among objects, as instances of separate classes. Classes expose events as well as methods in their interfaces. Objects announce events to notify registered observers of occurrences. Mediator instances function as observers that effect component integration upon notification. Events are thus like join points, and mediators like aspects. The mediator style has no equivalent of a pointcut sublanguage, however.

Yet what we found was that the apparently strictly more expressive AspectJ did not adequately support instance-level integration. An aspect, serving as a mediator, influences *all* instances of all advised types. Nor did AspectJ’s limited model of aspect instances help. First, the programmer has no control over aspect instantiation, which is mandatory per object and handled entirely by the AspectJ runtime. Second, an aspect instance can only advise exactly one object, and mediation essentially involves advising multiple objects of different types.

The type-orientation of AspectJ thus drove us to a design strategy in which a single aspect instance acts as a proxy for, and emulates, multiple instance-level aspects. This global aspect instance is invoked by triggering actions taken by any instance of any advised object. This is the source of unnecessary overhead at both design time and runtime. At design time, aspects modules must be programmed to emulate multiple aspect instances. At runtime, actions on an object of any advised type *requires* that aspect code be invoked to determine, at a minimum, whether the affected object is an *advised* instance. All objects of all advised types are impacted, and the impact grows with the number of aspects advising a given type.

This structure has several consequences in terms of design complexity and runtime overhead. First, it throws the designer into a pre-object-oriented style of design where abstract data type modules manage all instances explicitly. Second, advice method invocation overhead will be incurred for all objects of a type that *might* participate in a given relationship, whereas, in the mediator style, invocation overhead is limited to participating instances, modulo a single very low cost check, at each event announcement point, for a non-zero number of registered listeners. Compiler optimizations might conceivably eliminate some overhead, but the ability for aspects instances to advise and unadvised objects dynamically makes deep analysis intractable.

A second investigation—an empirical style of aspect-oriented program design and evolution by Bill Griswold and his students at the University of California, San Diego, further showed the issue with current languages. Griswold’s group reengineered his *AspectBrowser* tool, replacing a naïve model-view-controller design-pattern

implementation. They used AspectJ to achieve a modular implementation. They found that of the 54 aspects used across 15,000 lines of code, 11 (20%) employed the “proxy” work-around described above, in with a single aspect emulating multiple aspect instances. The resulting design complexity is managed by pattern-like stylized use and naming conventions to signal dispersed components’ roles.

A third investigation pointed to the performance impact of type-level-only aspects. We did an exploratory study of the performance penalties associated with having aspects advise types in AspectJ and HyperJ. The results are presented in Fig. 2. The *X* axis shows the number of aspects advising a simple *Bit* type, having methods to get and set a one-bit value. The *Y* axis shows the cost of invoking a *Bit* member function a fixed, large number of times. The cost per invocation clearly rises with the number of aspects advising the type—and will do so for every instance of the type. The advice code in this case did nothing but return immediately. In practice, the minimum function of much advice, as we have observed, is to check to see whether the invoking object is one that needs to have something done, further increasing the costs, typically by the cost of a hash table lookup.

By contrast, as the chart shows, a mediator style of integration imposes a very small, constant overhead. Hooks need to be present in the Bit objects to notify registered mediator instances, but if none are registered, the cost is constant (a single *if* statement). It does not matter how many mediator types can “advise” a bit, but only how many mediator instances actually do so.

The challenge is to learn how to reconcile the provision of rich and expressive join point models and pointcut languages with the need for aspects at the instance level. We thus propose to explore the whole question of instance-level aspects in aspect-oriented languages that transcend traditional wrapper-based architectural styles. We aim to develop and evaluate language mechanisms that preserve the benefits of rich aspect languages while providing needed support at the instance level.

4. Approach and prototype: *ilaC#*

A key element of the approach that we are taking to begin our exploration of these issues is to statically analyze aspect-oriented programs to determine what join points *might* be advised by aspect instances. We use the analysis results to selectively and automatically instrument those join points with event notification code that supports runtime registration of aspect instances. Thus, we intend to use a traditional implicit invocation structure as a runtime mechanism, but to abstract it from the language definition behind a rich pointcut language. We plan to investigate opportunities for additional optimizations in the future.

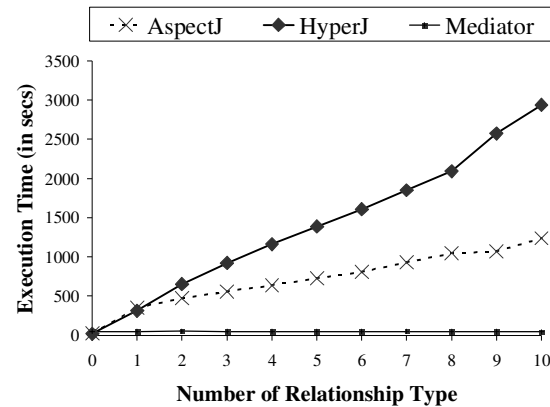


Fig 2: Performance Curves for AspectJ, HyperJ and Mediator based Design.

In order to test the feasibility of the idea and to have an experimental system for research in the area, we have begun to develop a new language and implementation, *ilaC#* (for *instance level aspects for C#*), pronounced self-deprecatingly as *i-lac-sharp*. Our goal is to make *ilaC#* an instance level aspect oriented language with a rich pointcut language based on C# [21]. We have already implemented a rudimentary prototype *ilaC#* weaver using the Code Document Object Model (CodeDOM) [23]. We hope to make a more mature but still fairly spartan prototype available in Spring of 2003.

For discussion purposes, we will be using the Bit example taken from our earlier work [26]. The example is as follows: Suppose that you are asked to construct a system out of several binary digit (Bit) objects. The system has *n* Bit instances, *b1 b2, b3, b4 ... bn*, each accessible to clients. The state space of a Bit is a single Boolean digit. Two mutator operations are applicable to a Bit, Set and Clear. An observer, Get, returns the current state.

There are numerous types of integration relationships possible between bits, and a given bit instance can participate in zero or more relationship instances. One such relationships type is *Equality*. An instance *e* of this relationship makes two Bit instances *b1* and *b2* work together so that whenever a client Set’s either Bit, the other is Set, too, before the first operation completes; and if a client Clear’s either bit, the other is Clear’ed, too, before the original operation completes. In other words *e* works to keep *b1* and *b2* in consistent states.

Trigger is a slightly different relationship. Bits *b1* and *b2* in a trigger relationship *t* behave such that if a client Set’s *b1*, then *b2* is Set before the original operation completes. Bit *b2* can be Set and Clear’ed with no effect on the other Bit’s, and *b1* can be Clear’ed with no effect on *b2*, but if *b1* is Set again, *b2* has to be Set, too.

Numerous relationship instances, *e1, ..., en, t1, ..., tm*, can exist in a given system. Moreover, a Bit *bi* can

participate in any number of instances of any number of relationship types. The structure of such a system is thus a network of Bit objects connected by relationship arcs.

ilaC# extends C# syntax by adding AspectJ keywords and capabilities, including *aspect*, *pointcut*, *advise*. It also adds a keyword *instancelevel* to modify aspects to specify instance level weaving. By declaring an aspect *instancelevel*, the developer tells the weaver to wait until runtime to weave aspect at join points. At compile time, the weaver attaches stubs at join points specified in the pointcut declaration, to enable runtime weaving at instance level. Here the pointcut declaration serves as a hint to the compiler attach stubs only to join points that *might* be advised at runtime. The importance of this strategy grows with the richness of the join point model.

In our simple, aspects declared *instancelevel* have implicit methods *addObject* and *removeObject* defined. They are similar to AspectJ version 0.6 *addObject* and *removeObject*. At runtime, these methods can be called with arguments specifying objects to be advised. A richer model is almost certainly needed as we go forward. There are two possibilities that we plan to explore: pointcut expressions implicitly pick out object instances to advise, and aspect instance constructors take as parameters the objects to advise.

```
public class Bit {
    bool value;
    public Bit() { value = false; }
    public void Set() { value = true; }
    public bool Get () {return value;}
    public void Clr() {value= false;}
}

public class TestHarness {
    public static void Main (string[] args){
        Bit bit1 = new Bit ();
        Bit bit2 = new Bit ();
        Equality eq = new Equality(bit1, bit2);
        bit1.Set(); bit2.Clr();
    }

    public instancelevel aspect Equality {
        Bit Bit1, Bit2;
        Equality(Bit B1, Bit B2) {
            Bit1 = B1; Bit2 = B2;
            addObject(Bit1);
            addObject(Bit2);
        }
        pointcut callSet(): call(void Bit.Set());
        pointcut callClr(): call(void Bit.Clr());
        after():callSet(){
            if(Bit1==(Bit)
                thisJoinPoint.getTarget())Bit2.Set();
            else Bit1.Set();
        }
        after():callClr(){
            if(Bit1==(Bit)
                thisJoinPoint.getTarget())Bit2.Clear();
            else Bit1.Clear();
        }
    }
}
```

Fig. 3 Implementation of Bit System using ilaC#

For now, we present the simple model. Consider the code in Fig. 3: (Our Bit Example) New keywords are shown in bold face fonts. The source code at a first glance looks like AspectJ code with *instancelevel* modifier and implicit *addObject* code method. The "Equality" method of the aspect in Fig. 3 can be considered as a constructor for the aspect. Weaving at an instance level is performed in this method by calling implicit method *addObject* on its two arguments. ilaC# allows this kind of instantiation for instance level aspects (but not for type level). This means that in order to put two bits in a relation, the application code needs to invoke the constructor explicitly on these two objects, thus introducing dependency between the base program and the aspect, which might be considered as a limitation of our current prototype.

We are investigating several options to overcome this limitation. We are investigating ways to implicitly identify which instances should be woven, e.g. weave an instance if it has some attribute set, or weave an instance if it is from code line # m to code line # n, or weave all instances that are constructed within a given scope, etc.

The modifier *instancelevel* when applied to the keyword pointcut advises the weaver to wait until runtime before weaving the aspect to the join points specified in the pointcut. As in the case of *instancelevel* aspects at compile time weaver will attach stubs to enable runtime weaving of aspects, but the stubs will only attached to the join points specified in this pointcut declaration. Applying the modifier to pointcut will make it possible to declare a combination of type-level and instance-level pointcuts in an aspect.

The pointcut language implemented by ilaC# is a subset of AspectJ, and we are expanding it further. We are also exploring a new pattern to specify join points by expanding the existing pointcut pattern class.method to class.{instance set}.method.

5. Related Work

Instance level customization dictates the need for instance level aspects. The idea of customizing instances according to their context is not new. The idea of roles and role models [3][11][19] is to customize objects to fit their roles, as determined by their context.

Kendall in [14][15] showed the implementation of role models based on an earlier version of the AspectJ tutorial [4]. Kendall [14] recognizes that aspect instances can serve to selectively extend object instance behavior based on the particular role an object plays in a system, but uses only the limited features of AspectJ (*per this, per target*). That work does not attempt to generalize, e.g., to the notion that an aspect might abstract behavior that cuts across several objects, based

on their collaboration relationships in a particular situation, or to move beyond the limits of AspectJ.

In the later work [15], addition of Aspects to objects on an instance basis is shown using the method *addObject* provided by AspectJ. The *addObject* method allows the programmer to attach an aspect to an object/instance as opposed to type, thus facilitating customization of objects according to context.

The methods *addObject* and *removeObject* were present in the AspectJ language until version 0.6, and were later removed from the language design in version 0.7 [5], so the implementation of role models using AspectJ became technically infeasible. There are discussion threads on applying aspects to objects/instances instead of types (or on the need for the *addObject* method) in AspectJ user's mailing list [5]. A few concrete examples are presented representing the need for context specific customization of objects using aspects (For detailed examples provided by Daniel De Luca, Jilles van Gurp, etc., see [5]).

The idea of instance-level aspects itself is not new. A prominent set of examples is in the class of "wrapper-based" aspect approaches, in which messages sent to and from components are intercepted for additional processing by aspect/wrapper objects. This idea is old and has appeared in many forms and contexts over the years: from the before and after methods of Common Lisp to tool integration frameworks, and so forth. It has been picked up and given an AOSD interpretation by efforts such as Sina/st[18][2] and ility-insertion [13]. However, these system designs assume a limited form of join point: namely message sending between objects. Certain strands of the instance-level constructs of early AspectJ are still present in the current language, namely in *per this* and *per target* constructs. However, the unresolved conceptual issues were daunting enough that the constructs were dropped for rapid deployment and low conceptual hurdles to language adoption [12]. The remaining constructs are irregular and not sufficient to address the problem in a general way. For example, there is no way to associate an aspect instance in AspectJ with *two* object instances of different types, e.g., to integrate their behaviors [24][25][26].

Nor is the idea of aspect C# new. AspectC# [17] is an AOSD implementation for C#. It enables weaving of type-level aspects without language extension. Cross Language Aspect Weaving (CLAW) [20] borrows many features from AspectJ. It contains an Execution Engine that weaves base and aspect code at runtime at the type level. AOP# [1] developed at Siemens Corporation aims at type-level extensions without language extensions using an AOP environment. Other approaches are Aspect.Net [22] and Aspect-Oriented Infrastructure for a Typed, Stack-based Intermediate Assembly Language [8]. We use C# for its merits as tool but our contribution

is in exploring the subspace of languages with instance weaving and rich pointcut sublanguages.

5. Conclusion

We have characterized aspect languages as falling into four subspaces. We recognized a void in "quadrant III." We have argued that there is sufficient reason to invest effort in exploring this quadrant: of aspect-oriented languages that combine rich pointcut sublanguages with instance-level advice constructs. We have committed to such an exploration. We are addressing the technical feasibility, costs, benefits, and relevant core issues.

We have adopted an implicit invocation runtime layer and compile-time instrumentation of potentially advised join points as an implementation strategy, recognizing there will be opportunities to optimize in some cases. We have also described an early prototype design and implementation based on C#.

The mediator style of design has been shown to have the potential to significantly ease the design and evolution of integrated systems [25] through a sharp separation of behavioral entity and relationship concerns and the composition of systems as networks of entity instances integrated by relationship instances. Aspect languages such as AspectJ bring rich pointcut sublanguages to the table, but in the process largely dropped instance-level composition capabilities. The work reported here promises to further ease software design and evolution by realizing the benefits of both approaches with added new synergies, at the cost of possibly significant increases in the complexity of language design and use.

Acknowledgements

This work was supported in part by the National Science Foundation under grant ITR-0086003. We thank Bill Griswold and his research group for their empirical data on instance-level aspect emulation in their AspectBrowser software tool.

References

- [1] AOP#: currently under development at Siemens Corporation. Egon.Wuchner@mchp.siemens.de.
- [2] 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.
- [3] Andersen, E. (Egil), Conceptual Modelling of Objects: A Role Modelling Approach, PhD Thesis, University of Oslo, 1997.
- [4] AspectJ: www.eclipse.org/AspectJ
- [5] AspectJ users mailing list archives can be obtained from www.eclipse.org/AspectJ.

- [6] L. Bergmans, The Composition Filters Object Model, Dept. of Computer Science, University of Twente, 1994.
- [7] Constantinos A. Constantinides and Tzilla Elrad, "Composing Concerns with a Framework Approach", Proc. 2nd Int'l Workshop on Aspect Oriented Programming for Distributed Computing Systems (ICDCS-2002), Vol. 2, jul, 2002, Vienna pp. 133-140.
- [8] Dechow, D., "Ph.D Dissertation Proposal", URL: <http://cs.oregonstate.edu/~dechow/>.
- [9] R. Douence, M. Südholt, "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.
- [10] Griswold, W.G., personal communication with Kevin Sullivan.
- [11] 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.
- [12] Hugunin, J., Personal Communication with Kevin Sullivan, OOPSLA Conference, 2002.
- [13] Filman, R. E. and Barrett, S. and Lee, D. D. and Linden, T., "Inserting Ilities by Controlling Communications", Communications of ACM, vol. 45, number 1, Jan, 2002, pp. 116-122
- [14] 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
- [15] Kendall, E. A., "Aspect-oriented Programming in AspectJ," Evolve 2000, Sydney, March, 2000
- [16] 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
- [17] Kim, H., "AspectC#: An AOSD implementation for C#", Technical Report TCD-CS-2002-55, Department of Computer Science, Trinity College, Dublin, 2002.
- [18] Koopmans, P.S., "Sina/St: User's Guide and Reference Manual", TRESE project, University of Twente, 1996.
- [19] Kristensen, B. B., "Object-oriented Modelling with Roles", OOIS'95, Proceedings of the 2nd International Conference on Object-oriented Information Systems, Dublin, Ireland, 1996.
- [20] Lam, J., "CLAW" URL: www.iunknown.com.
- [21] Microsoft. C# Specification Homepage. <http://msdn.microsoft.com/net/ecma/>.
- [22] Microsoft Aspect.Net project description. <http://research.microsoft.com/programs/europe/rotor/Projects.asp>.
- [23] Microsoft .Net Framework Developers Guide available at <http://msdn.microsoft.com>
- [24] Sullivan, K., "Mediators: Easing the Design and Evolution of Integrated Systems", Ph.D. dissertation, University of Washington, 1994.
- [25] 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).
- [26] 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
- [27] Tarr, P. and Osher, H., "Hyper/J™ User and Installation Manual", IBM Corporation.