# One More Step in the Direction of Modularized Integration Concerns

Hridesh Rajan
*University of Virginia*
*hr2j@cs.virginia.edu*

## 1. Research area

Component integration, ease of design and evolution of integrated systems.

## 2. Problem statement

Component integration creates value by automating the costly and error-prone task of imposing desired behavioral relationships on components manually. Requirements for component integration, however, complicate software design and evolution in several ways: first, they lead to coupling among components; second, the code that implements various integration concerns in a system is often scattered over and tangled with the code implementing the component behaviors. Straightforward software design techniques map integration requirements to scattered and tangled code, compromising modularity in ways that dramatically increase development and maintenance costs.

## 3. Prior research

Designing integrated systems using simple object oriented techniques requires components to refer to other components with which they are integrated, resulting in a *names* relationship with the other component. Components to observe the desired behavior will need to *invoke* each other, which will be achieved by calling each other and thus there will a *name* dependence between these components resulting in coupling and preventing separate compilation, link, test, use, etc.. Integration concern is scattered and tangled across the components resulting in code complexity and non-modularity in design.

Implicit invocation techniques [3], e.g. subject-observer pattern, allow better management of *names* relationship. In this design technique, observers register with subjects that in turn implicitly invoke them without naming them. Observer still *names* and *invokes* the subject. In addition, the integration concern is still scattered and tangled across the components.

The mediator-based design approach [12][13] was developed to enable the modular representation of behavioral relationships to ease component integration. The *Behavioral relationship* is defined as a protocol for coordinating the control, actions, and states of subsets of system components to satisfy part of the integration requirements for the system. Integration concern is largely modularized and represented as object-oriented mediator classes. Integration is achieved by declaring events as part of the component's interface. Mediators then register with these exposed events to receive notifications to create the required invocation relations from components to mediators without inducing *names* dependences, however, the event declaration, announcement, and registration code, which is related to the integration concern, is still scattered across the component. Further, mediator requirements dictate the need for events declared and announced by a component.

## 4. Research hypothesis

Recent aspect-oriented [7] techniques 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 such techniques is a modular representation of a *crosscutting concern*, while a mediator is a modular representation of a behavioral relationship (integration concern), which can be seen as a particular kind of crosscutting concern.

AO methods thus suggest an improvement on the existing state of the art in component integration. 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.

As described before 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 research hypothesis 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.

## 5. Solution approach

To ease the design and evolution of integrated systems, mapping of the mediator approach into the design space of AspectJ [1] was attempted. The results [10][14] were encouraging but mixed and revealed some shortcomings of the *AspectJ* design with respect to its usability in this context. The language does not 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 and design costs.

Another disadvantage of AspectJ-like languages 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* [15], an integrated environment for radiation treatment planning—itself a major test of the mediator approach, such events were routine. *AspectJ*-like languages do not expose such events as join points.

In order to map mediators to aspects in a completely satisfactory way, current language model of AspectJ-like languages needs to be generalized in the following dimensions: first, support for *instance-level advising* and *first class aspect instances* needs to be added, and second the join point model needs to be extended to expose a far wider set of execution phenomena. 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.

Expanding the join point model beyond join points anchored to the interface elements raises some issues [9] regarding the stability of the reference to the join points and the degree of unpredictability that will result from incorporating a wider set of execution phenomenon as join points. There are similar concerns for reasoning about implicit invocation and there has been some work in this direction [2], [4], [5], and [6]. We aim to exploit the mapping from implicit invocation space to aspect-oriented space and existing body of knowledge on reasoning about implicit invocation to enable reasoning about aspect-oriented programming in general and the fine-grained join point model provided by our work in particular.

## 6. Contributions

This research will make the following contributions:

I. Language model of the AspectJ-like languages will be extended with first-class aspect instances, instance-level advising and finer-grained join point model,
II. Proof of concept that the resulting model supports a full fledged, aspect-oriented variant of mediator-based design that relieves developers of the need for explicit event declaration, announcement, and registration,
III. Mapping from implicit invocation space to aspect-oriented programming space and utilization of the existing body of knowledge in reasoning about the implicit invocation will enable reasoning about aspect-oriented programming,
IV. Further modularization of integration concerns by enabling component integration without requiring any change in components.

## 7. Evaluation

To evaluate the claims we are implementing an AspectJ-like extension to C# [8] language called Eos [11]. The *Eos* compiler supports the complete C# language as well as AspectJ-like constructs, instance-level aspects. We are currently analyzing the tradeoffs associated with expanding the join point model and defining appropriate pointcut expressions to be used for selecting these new join points.

To test the hypothesis that *Eos* supports the design of realistic systems using aspect instances as mediators, we have already implemented, in *Eos,* key mediator structures used in the design of *Prism*. This initial implementation of these mediator structures revealed shortcomings of the join point model and the need for exposing more type of events as join points. We will be revisiting these structures once Eos is equipped with a fine-grained join point model.

In addition to *Prism*, we are also experiencing real needs for component integration in the Eos compiler itself and we will use it as a second case study of our approach. We are also exploring the open source projects available for potential case studies. Eos compiler is available for research and teaching purposes. The use of compiler for experimental and real world projects might lead to more case studies.

## 8. References


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

[2] Bradbury, J., and Dingel, J., "Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems". European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'03). Helsinki, Finland. September 2003.

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

[4] Dingel, J., Garlan, D., Jha, S., and Notkin, D., "Reasoning about Implicit Invocation", Proceedings of the Sixth International Symposium on the Foundations of Software Engineering (FSE-6), Lake Buena Vista, FL, November 1998.

[5] Garlan, D., and Khersonsky, S., "Model checking implicit-invocation systems.", In Proc. of the 10th Int'l Workshop on Software Specification and Design, Nov 2000.

[6] Garlan, D., Khersonsky, S., and Kim, J. S., "Model checking publish-subscribe systems." In Proc. of the 10th Int'l SPIN Workshop on Model Checking of Software, May 2003.

[7] 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.

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

[9] Ossher, H., and Tarr, P., "Operation-Level Composition: A Case in (Join) Point", Workshop on aspect-oriented programming, ECOOP 1998.

[10] Rajan, H., and Sullivan, K., "Need for Instance Level Aspects with Rich Pointcut Language", In the proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies (SPLAT) held in conjunction with AOSD 2003, Boston, MA, USA, Mar 2003.

[11] Rajan, H., and Sullivan, K., "Eos: Instance-Level Aspects for Integrated System Design", 2003 Joint *European Software Engineering Conference* and ACM SIGSOFT Symposium on the *Foundations of Software Engineering* (ESEC/FSE 03), Helsinki, Finland, Sept 2003.

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

[13] 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).

[14] 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

[15] 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.