Aspect Reasoning by Reduction to Implicit Invocation

Jia Xu
Department of Computer Science, University of Virginia
151 Engineer's Way, P.O. Box 400740
Charlottesville, Virginia 22904-4740, USA
+1 434 982 2296
jx9n@cs.virginia.edu

Hridesh Rajan
Department 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
Department 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
Aspect-oriented programming constructs complicate reasoning about program behavior. Our position is that we can *reduce* key elements of aspect programming to implicit invocation (II) and then use existing work on reasoning about II to reason formally about aspect programs. We map aspect-oriented programs to equivalent programs with join points and advice replaced by event notifications and observers; use existing techniques for reasoning about programs that use implicit invocation; and then interpret the results in the context of the original aspect-oriented program.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Constructs and Features – *abstract data types, polymorphism, control structures.*

General Terms
Measurement, Design, Experimentation, Languages, Verification.

Keywords
AOP, Implicit Invocation, Reasoning, Model Checking

1. INTRODUCTION
The ability of aspect-oriented [15][16] approaches to enable modular representation of crosscutting concern by implicit behavioral modifications at join points specified by predicates on program elements is a bane for reasoning. In the presence of aspects, the behavior of a module at runtime can not be determined by just looking at the module code. One is required to understand the possible effects of each aspect in the system. The actual behavior is determined by composing the base and aspect behaviors. It is widely understood that reasoning about AOP remains a challenge [5]. Our position is that a reduction from the space of aspect-oriented programs to the space of programs using implicit invocation has the potential to enable formal reasoning about properties of aspect-oriented programs using existing methods for reasoning about implicit invocation systems.

The problem is well known. Several approaches have been proposed to enable automated reasoning about aspect-oriented programs. Some of these approaches try to apply model checking to verify properties of aspect-oriented programs [3][19][24], while others try to reduce aspect-oriented programming model to simpler models which can be easier to reason about [1][5][6][7].

Our contribution is in seeing how to exploit the relationship between join points and events in implicit invocation systems [12]. In such systems, modules expose events, with which other modules register procedures. Registered procedures are invoked when modules announce events, extending the modules' behaviors implicitly. Implicit invocation is widely used for complex system design. Sullivan and Notkin [23] showed how implicit invocation enables separation of integration concerns to ease the design and evolution of integrated systems and how it poses AOP-like problems in reasoning about II systems.

The problem of reasoning about implicit invocation (II) has generated significant interest over the last decade. In particular, Garlan et al. [11] proposed an event model to describe the behavior of the II systems. They then use model checker to check property assertions on this event model. Bradbury et al [4] further refined Garlan et al.'s approach and evaluated their approach in real world software systems, demonstrating the feasibility of applying formal reasoning techniques to real II systems.

Our position is that reducing the join point and advice model of aspect programming to II is possible, as shown by Eos [18], and that this reduction permits formal reasoning techniques for II systems to be applied to aspect programs. We first map an aspect-oriented program that uses join point and advice to a semantically equivalent implicit invocation program; we reason about it using existing techniques; we then map the results from the II space back to the AOP space. In our earlier work [18], we showed that the implicit invocation space can be mapped to the aspect space. (In particular, support for instance-level aspects and first class aspect instances enables a mapping of aspect programs to mediator-based design structures [21][23], which use implicit invocation extensively to separate integration concerns.) In this work, we make the reduction concrete and present preliminary evidence supporting our hypothesis.

In the rest of this paper, we will be using an example system from our previous work [18], [22] to illustrate various approaches. The example is extremely simple, but it is known to capture essential issues in a way that scales up. Our example system consists of two objects *b1* and *b2*, instances of the *Bit* type. A *Bit* can be Set and Cleared by *Set* and *Clear* and its current state can be read by the *Get* method. In our example system *b1* and *b2* are required to work together as follows: if any client *Sets* (respectively *Clears*) either *Bit*, the other must be *Set* (*Cleared*). In other words, the behaviors of the *Bits* have to be integrated by a behavioral relationship, which we will call *Equality*, which maintains a bit-equality constraint.

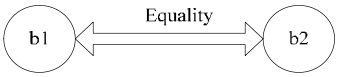

Figure 1. A simple example: Bit

The rest of this paper is organized as follows. Section 2 describes aspect oriented programming and the challenges in reasoning about it. Section 3 describes the effort in reasoning about implicit invocation space. Section 4 describes our approach. Section 5 presents related work. Section 6 concludes.

2. ASPECT-ORIENTED PROGRAMMING

Aspect-oriented programming constructs are meant to enable the modular representation of otherwise scattered and tangled code. The key mechanisms of aspect-oriented programming in the tradition of AspectJ [2] are join points, pointcut expressions, advice code, and aspect modules. A *join point* is a point in the execution of a program (such as just before a method body executes) exposed by the language design for behavioral modification by aspect modules. The join points exposed by current AspectJ-like languages include method calls and execution, field get and set operations, exceptions, and object initialization. A *pointcut* is an expression—a predicate—that serves to select a subset of program join points. *Advice* is code that is effectively to be executed at each join point selected by a pointcut. An *aspect* is a module that aggregates pointcut expressions and associated advice code along with other information typically found in class definitions. Weaving is the process by which advice code is composed with the base program code at selected join points to yield an executable.

Eos [13] [18] is an AspectJ-like extension of C# [17] that supports first-class aspect instances and instance-level advising. By first class aspect instance we mean that the aspects are class-like constructs that can be instantiated, passed as arguments, returned as value, etc. By instance-level advising we mean ability to select specific *instances* of a type that will be affected by aspect advice. (In AspectJ, aspects advise types and thus all instances.) The code for the Bit example in Eos is as follows:

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

The following code implements Equality as an instance-level aspect:

```
1  public instancelevel aspect Equality  {
2    Bit b1, b2;
3    bool busy;
4    public Equality(Bit b1, Bit b2)   {
5              addObject(b1);  addObject(b2);
6              this.b1 = b1;    this.b2 = b2;
7              busy = false;
8   }
9   after():execution(public void Bit.Set ())   {
10      if(!busy) {
11             busy = true;
12             Bit m = (Bit) thisJoinPoint.getTarget();
13             if(b == m1)b2.Set(); else b1.Set();
14             busy = false;
15             }
16   }
17   after():execution(public void Bit.Clear ())   {
18      if(!busy) {
19             busy = true;
20             Bit b = (Bit) thisJoinPoint.getTarget();
21             if(b == b1)b2.Set(); else b1.Set();
22             busy = false;
23             }
24   }
25  }
```

Figure 2. Eos code for Bit example

The purpose of the aspect is to ensure that b1 and b2 always have the same state at quiescent points (i.e., except during execution of a Set operation). We thus need to verify that the aspect module behaves in such a way. It is, however, generally difficult to reason about AOP for the following reasons:

1. The primitive constructs in aspect-oriented languages need to be rigorously defined.
2. It could be very hard to reason about an aspect program automatically. There has been a fair amount of research on the possibility of applying model checking on reasoning about AOP, although there is hardly a working example.
3. Since the behavioral modifications by aspects can cut across the entire code base, it's very hard for us to understand an aspect-oriented program in a modular way. That is, we can no longer analyze modules separately then combine results. An aspect can influence the semantics of the whole system. This issue as the most difficult part of reasoning about AOP.

3. IMPLICIT INVOCATION

Implicit invocation [23] [12] is a mechanism for managing how *invocation* relations are represented as *names* relations. If component *A* needs to *invoke* component *B* at a certain point, *A* can do so either by explicitly calling *B*, in which case *A names* B, or *B* can *register* with *A* to be invoked implicitly by event announcement, which case, *B* names *A.* Because the *names* relation is a key determinant of compile, link, and runtime dependencies, having means to structure it properly is important. Implicit invocation and join points and advice provide such means.

II is also known as publish-subscribe system, since generally it is implemented in such a way. A component (the subscriber) registers interest in particular events that the other component (the publisher) announces. The II mechanism then guarantees the invocation of subscriber. The publisher is not aware of the existence of the subscribers. II has been used widely in system-level development and message-passing applications. For example, a user can define and register a callback procedure that is invoked when a particular signal is raised by the OS kernel.

Such systems make modular reasoning harder, since we need to decouple the verification of one component from the verification of the rest of system that communicate with the given component by event bindings. Dingel et al. [8] proposed a formal model for II systems and proposed a three-phase reasoning methodology: decomposition, local reasoning and global reasoning. By the decomposition process, we can formalize the event/handler semantics and model the system, the environment and the event dispatch mechanism in a modular way. Applying the three-phase reasoning then can be expected to achieve the effect of modular reasoning about the whole system.

It is, however, often not easy to decompose the system into separate groups and prove their independence. An alternative approach proposed by Garlan et al. [11] uses model checking

instead of formal modular reasoning. Application of model checking to software encounters two problems. First, an appropriate state model for the system being checked needs to be created. This state model of a reasonable size system has a huge state space. To check this huge state space using a model checker is time consuming, if not infeasible. Second problem is thus to find the means to reduce this state space into manageable size so that it can be supplied as an input to the model checker.

The architecture of an II system in [11] models the following features:

- Components: functional objects with well-defined interfaces
- Events: the primary communication method between components
- Event-Method Bindings: the correspondence between announced events and the methods that are invoked in response as event handlers
- Event Delivery Policy: rules about event announcement and delivery
- Shared State: another communication method between elements of the II system
- Concurrency Model: determines if the system has a single thread or multiple threads of control

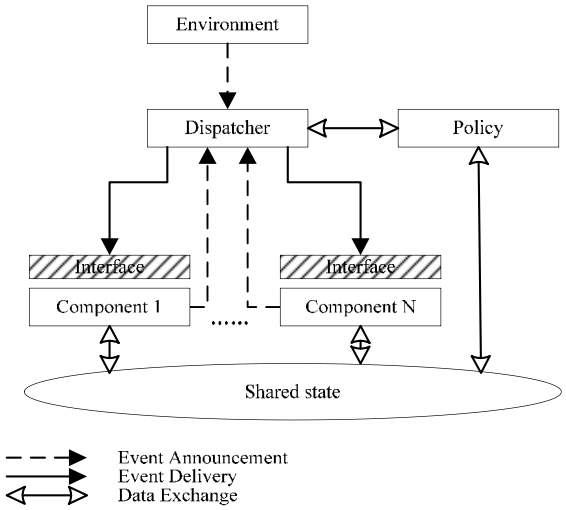

Figure 3. II system architecture

The system structure is in Figure 3. Besides those functional components, the dispatcher and policy modules are another important part of the approach. They are responsible for event binding and dispatching. Also the environment represents external elements that could affect the system.

The run time state model of an II system has to model the following in addition:

- Event announcement by the system components
- Storage of event announcements before dispatching
- Event delivery to the system components
- Invocation of methods bound to the delivered events
- Invocation acknowledgement

For the *Bit* example, *b1* and *b2* are considered separate components. Calling the state change methods *Set*/*Clear* on the component *b1* results in the component announcing an event representing the change in its state, which is then captured by the dispatcher. The dispatcher will consult the policy module to determine what event will be delivered to which component without causing a propagation cycle. The state change in b2 will also result in announcement of an event representing its state change and the same actions as above. Figure 4 depicts the simplest II state model that models the *Bit* example, in which we omit the environment module and the details of b2's event exchange since it's the same with b1. For each event of interest, a notify message is delivered from the component to the dispatcher, which results in the delivery of an invoke message from the dispatcher to the other component.

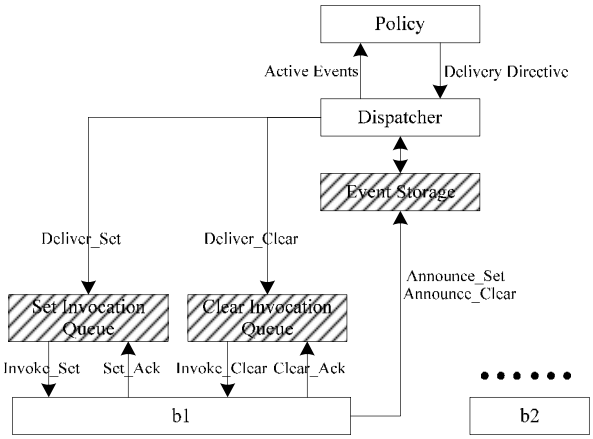

Figure 4. II run time state model for the Bit example

To reason about the system, we are interested in verifying some properties of the system. These properties are expressed as assertions. The *Equality* between *b1* and *b2* will be represented as the following assertion:

Equality:
assert(F (b1.state = b2.state));

"F", a logical notation of LTL, represents "eventually". The SMV model checker takes these assertions and the state machine constructed before as input and produces validity of these assertions as output.

Bradbury et al. [4] extended Garlan et al.'s model to support dynamic event model. They use XML to represent the event model which is later translated into SMV input. They have applied their method on real world implicit invocation systems such as Active Badge Location System (ABLS) [26] and Unmanned Vehicle Control System (UVCS) [20] demonstrating the capabilities of their approach.

4. Reduction from AOP to II

We would like to assure that the aspect performs its intended behavioral modifications without producing any undesirable side effect. Existing approaches can be used to reason about the plain object-oriented systems. Our approach therefore focuses on reasoning about aspects, the modular representation of crosscutting concerns, and its interaction with the component part. To enable this reasoning, we propose to reduce an element of aspect-oriented programming space (an aspect-oriented program)

to an element in implicit invocation space (an implicit invocation based program). We can then reason about the element in the implicit invocation space using the II reasoning techniques described in the last section. The reasoning results will then be reduced back from II space to the AOP space thus effectively enabling reasoning about the aspect-oriented program.

First, let us revisit the concepts of aspect-oriented programming as embodied in asymmetric languages such as AspectJ. A *join point* is a point in the execution of a program (such as just before a method body executes) exposed by the language design for behavioral modification by aspect modules. The join points exposed by current AspectJ-like languages include method calls and execution, field get and set operations, exceptions, and object initialization. A *pointcut* is an expression—a predicate—that serves to select a subset of program join points. A *pointcut* is then defined to be a predicate expression over a set of join points.

We observe that every join point can be viewed as a set of semantic events. These events will be announced when the control hits the join point during program execution. AspectJ-like languages can advise a join point in three different ways: *before*, *after* and *around*. The before and after advice are semantically clean, however, the around advice is a bit more involved. To keep the model simple we will not be discussing around advices. To differentiate between these ways of advising we map each join point to a 2-tuple of events: <before the join point, after the join point>. We treat each of these elements differently in the corresponding event model. An advice before a join point is mapped to the event "before a join point" and similarly for after advice.

A pointcut selects a subset of program join points. Each pointcut is mapped to an enumeration of a set of 2-tuple of events where each 2-tuple in the set corresponds to a join point in the subset of program join points selected by the pointcut. A named pointcut is mapped to a named set of 2-tuples. The subset of join points that are matched by the pointcut expressions that rely on run-time information cannot be obtained statically. The control flow (*cflow*) and control flow below (*cflowbelow*) are examples of some of these pointcuts. These pointcuts cannot be mapped statically to a simple event model like the one used by Garlan [11]. In summary, the mapping of join points and pointcuts to II concepts can be shown below:

f: {Joinpoint} -> {Event} X {Event}

g: {Pointcut} -> {Predicate} X {A set of events}

A pointcut can be denoted by a pair *<predicate, {joinpoint}>*, which means a predicate expression over a set of joinpoints.

For example, the mapping applied to a joinpoint *a* is:

f(a) = <after_a, before_a>

in which *after_a* and *before_a* are two events.

The mapping applied to a pointcut <p, {a, b, c}> is:

g(<p, {a, b, c}>) = <g(p), {f(a), f(b), f(c)}>

in which *p* is a predicate over the set of joinpoints *a*, *b*, *c*, while *g(p)* denotes the mapped predicate over the set of events.

Events picked out for our Bit example are shown below:

Events exposed by the Bit component =
{ [before_Bit.Bit, after_Bit.Bit],
[before_Bit.Set, after_Bit.Set],
[before_Bit.Clear, after_Bit.Clear] }

Events picked out by the pointcut expression "execution(public void Bit.Set ())" = {[before_Bit.Set, after_Bit.Set]}

Events picked out by the pointcut expression "execution(public void Bit.Clear ())" = {[before_Bit.Clear, after_Bit.Clear]}

The mapping of pointcut *after(): execution(Bit.Set())* is:

g(<after execution, {Bit.Set()}> =

<only after_* events, {<before_Bit.Set, after_Bit.Set>}>

The mapping of pointcut *after(): execution(Bit.Clear())* is:

g(<after execution, {Bit.Clear()}>) =

<only after_* events, {<before_Bit.Clear, after_Bit.Clear>}>

An advice is mapped to an event handler. The role of an advice, with respect to the advised pointcut, is the same as the event handler with respect to the captured event. In the Bit example, there are two event handlers in the *Equality* aspect, one corresponds to the after():execution(Bit.Set()) advice, the other corresponds to the after():execution(Bit.Clear()) advice.

Third, there should be a dispatcher in the mapped system, as well as a dispatch policy module. The dispatcher is responsible for event storage, event binding, event delivery and interacting with the dispatch policy module. The policy module implements event delivery policy. In the context of mapping AOP, it should be able to choose event handlers according to a predicate expression over a set of events, just like the pointcut definition.

As for the Bit example, the aspect *Equality* actually can be mapped to part of the policy module in Figure 3. When the dispatcher receives an event, it will inquire the policy module to decide the actions it will take. In this case, the *Equality* policy will determine which message (Set or Clear) to deliver to which component.

The assertions we can check over this II system like *assert(F (b1.state = b2.state))* is now mapped back to the behavior constraint between the two objects b1 and b2 in the Eos program (Figure 2). This constraint is therefore checked by the reduction process. Thus, we demonstrate a simple example of our approach to use II reasoning technique to reason about an aspect-oriented program's behavior.

5. RELATED WORK

Dingel et al. [8], Garlan et al. [11], and Bradbury et al.'s [4] work on model checking implicit invocation systems is closely related to ours—and is, in fact, used as a subroutine. They proposed an event model to describe the behavior of the II systems. Bradbury et al.'s approach translates this model written in XML format to the SMV language and applies the SMV model checker. These approaches are applicable to II systems, but not directly to aspect-oriented programs. Our approach supplements these approaches by providing a reduction from the AOP space to II space, thus enabling the use of these approaches in the AOP space.

Mapping AOP to event model is not a completely new idea. Filman and Havelund [10] briefly proposed an event language for aspects. The event language has primitive events and a set of relationships between events, which include abstracted temporal relationships, abstract temporal quantifiers, concrete temporal relationship referring to clock time, cardinality relationships and aggregation relationships for describing sets of events. Walker and Murphy [25] employed their implicit context concept to map join points to ordered events. By such mapping, they showed a close relationship between AOP and implicit context. Our work makes the reduction from AOP to II explicit.

As for reasoning about AOP, there has been significant research on this topic. Ubayashi et al. [24] claimed to apply model checking using aspects. They write an aspect for every property to check, and then weave these aspects and the source program into a new program and then execute this weaved program. This approach works only for plain java programs. It can only check properties that can be represented by aspects. It also uses a dynamic approach, so presence of property violation can only be discovered if that execution path is taken.

Blair and Monga [3] view every pointcut declaration as a *slicing criterion* that can be used to compute an associated slice. They then envision that this sliced program could be fed into Bandera model checker, but the expressiveness of aspects is difficult to be captured by any slicing technique.

Instead of reasoning about the entire program, Clifton and Leavens [5][6] give two concepts for AspectJ: Observer (Spectator) and Assistant. Assistants are aspects that could change the behavior of other parts, while observers do not. They also propose an *accept* notation to be added into AspectJ, to make aspect invocation explicit, for facilitating modular reasoning. By categorizing aspects into observers and assistants, and explicitly exposing the join point, they expect to be able to reason AOP in a modular way, however, it remains unclear how can we differentiate assistants from observers in real programs. The *accept* notation compromises the obliviousness [9] properties of aspect-oriented programs. Our approach on the other hand, does not impose any restriction on the language model of aspect-oriented programming languages.

Devereux [7] tries to transfer aspect programs to alternating-time logic. Then program properties can be expressed by assertions in alternating-time logic. It supports two concepts, imposition and preservation similar to assistant and observer. The development of a reduction similar to ours from aspect-oriented space to alternative-time logic is possible; however, the lack of tool support for automated reasoning in alternating-time logic makes the reduction less attractive.

Recently there has been increasing research interests on exploiting type systems to enable reasoning about aspect-oriented programs. Aldrich [1] presented a simple aspect language called TinyAspect. Module sealing and explicit declaration of exported join points is the core of TinyAspect. The idea is to enforce abstraction by prohibiting clients, viz., aspects, from exploiting implementation details, such as calls from within a component to its own public methods. There is a set of type inference rules for TinyAspect by which one can reason about the behavior of aspects. Type checking in the TinyAspect model, however, does not allow one to reason about the kinds of behavioral properties that we address.

6. CONCLUSION

Aspect-oriented programming imposes many new challenges on program understanding and reasoning. In fact, how to reason about AOP in a modular way has been an open question for years. In this paper, we reduce the join point and pointcut mechanisms of AOP to the events of implicit invocation systems, and we show that this reduction has the potential to improve our ability to reason formally about the aspect program behavior. Forthcoming work will formalize the reduction, develop and evaluate the approach, and investigate the possibility of automated tool support for such reductions and formal property verifications.

7. ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under grant ITR-0086003.

8. REFERENCES

[1] Aldrich, J., “A Typed, Modular Foundation for Aspect-Oriented Programming”, 2003.

[2] AspectJ Homepage: http://www.eclipse.org/aspectj.

[3] Blair, L., Monga, M. “Reasoning on AspectJ Programmes”, GI-AOSDG 2003 Essen, Germany

[4] Bradbury, J. S., Dingel, J., “Evaluating and Improving the Automatic Analysis of Implicit Invocation Systems,” In Proc. of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on the Foundations of Software Engineering (ESEC/FSE 2003), Helsinki, Finland, Sept. 2003.

[5] Clifton, C., and Leavens, G. T., “Observers and Assistants: A Proposal for Modular Aspect-Oriented Reasoning.”, Technical Report TR#02-04, Department of Computer Science, Iowa State University, March 2002.

[6] Clifton, C., and Leavens, G. T., “Obliviousness, Modular Reasoning, and the Behavioral Subtyping Analogy.”, Technical Report TR#03-01A, Department of Computer Science, Iowa State University, March 2002.

[7] Devereux, B., “Compositional Reasoning About Aspects Using Alternating-time Logic”, FOAL2003

[8] Dingel, J., Garlan, D., Jha, S., Notkin, D., “Reasoning about implicit invocation”, Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering, November 1998

[9] Filman R., and Friedman, D., “Aspect-oriented programming is quantification and obliviousness”, In Proc. Workshop on Advanced Separation of Concerns, OOPSLA 2000.

[10] Filman, R.E., Havelund, K.. “Realizing Aspects by Transforming for Events.” In Automated Software Engineering 2002 (ASE'02). Edinburgh, Scottland, 23-27 September 2002. IEEE Computer Society

[11] Garlan, D., Khersonsky, S., and Kim, J. S., ”Model Checking Publish-Subscribe Systems”, Proceedings of The 10th International SPIN Workshop on Model Checking of Software (SPIN 03), Portland, Oregon, May 2003.

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

[13] Eos Homepage: http://www.cs.virginia.edu/~eos

[14] Java: http://java.sun.com

[15] 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.

[16] Masuhara, H., and Kiczales G., "Modular Crosscutting in Aspect-Oriented Mechanisms", ECOOP 2003, Darmstadt, Germany, July 2003.

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

[18] 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).

[19] Sihman, M. and Katz, S.. "Model Checking Applications of Aspects and Superimpositions", FOAL 2003

[20] Stuurman, S., and Katwijk, J.van, "On-line change mechanisms: the software architectural level", In Proc. Of the ACM SIGSOFT FSE, pages 80-86, Nov. 1998

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

[22] 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

[23] 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).

[24] Ubayashi, N., Tamai, T., "Aspect-oriented programming with model checking", Proceedings of the 1st international conference on Aspect-oriented software development, April 22-26, 2002, Enschede, The Netherlands

[25] Walker, R. J. and Murphy, G. C., "Joinpoints as ordered events: towards applying implicit context to aspect-orientation", Workshop on Advanced Separation of Concerns at the 23nd ICSE, 2001.

[26] Want, R., Hopper, A., Falcao, V., and Gibbons, J., "The active badge location system." ACM Trans. on software engineering and methodology, 10(1):91-102, Jan. 1992