Nu: Towards an Aspect-Oriented Invocation Mechanism

Technical Report, Dept. of Computer Sc., Iowa State University

Hridesh Rajan Robert Dyer Harish Narayanappa Youssef Hanna
Dept. of Computer Science
Iowa State University
226 Atanasoff Hall
Ames, IA, 50010, USA
{hridesh, rdyer, harish, ywhanna}@cs.iastate.edu

ABSTRACT

The contribution of this work is the design, implementation and evaluation of a new aspect-oriented invocation mechanism for preserving design modularity in object code. We call our mechanism *Bind*. We make three basic claims. First, it is feasible to realize a programming model that supports *Bind* to preserve design modularity in object code. Second, the new invocation mechanism further improves the conceptual integrity of the aspect-oriented programming models by allowing advising and runtime properties of aspect-like constructs to be modeled as simple combinations of invocation primitives as opposed to new language constructs. Third, it brings new possibility for structuring aspect-oriented systems, removing the commitment to a single aspect-language model, and expanding the program design space to include arbitrary combinations of language models and advising structures. To support these claims, we present the design and implementation of *Nu*, a programming model based on the .NET Framework that supports *Bind* as an invocation mechanism. We show that *Nu* supports aspect-oriented program designs where multiple aspect-language models can be emulated using *Bind*, and used in arbitrary combinations without compromising the design modularity in the object code.

Categories and Subject Descriptors

D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.3.3 [**Programming Languages**]: Language Constructs and Features —Control structures; Procedures, functions, and subroutines; D.3.4 [**Programming Languages**]: Processors — Code generation; Incremental compilers; Run-time environments

General Terms

Design, Human Factors, Languages

Keywords

Bind, Nu, invocation, incremental, weaving

1. INTRODUCTION

Today's aspect-oriented programming (AOP) languages [16, 24] provide software engineers with new possibilities for keeping conceptual concerns separate at the source code level [53, 13]. For a number of reasons[1], aspect weavers sacrifice this separation in transforming source to object code (and thus the very term *weaving*)(see Figure 1). The implementations of the crosscutting concerns are scattered and tangled again with the base code. The level of scattering and tangling in the object code varies with the underlying weaving approach. Some approaches instrument the code to insert calls at compile time [23], some only insert hooks at compile time so that at runtime observer like aspects can register with these hooks [47], yet another kind insert flags and/or meta information at compile time so that instrumentation can occur at load time [26], etc.

In this paper, we argue that sacrificing modularity has significant costs, especially in terms of the speed of incremental compilation and in testing. We argue that design modularity can be preserved through the mapping to object code, and that preserving it can have significant benefits in these dimensions.

The contribution of this work is a new aspect-oriented invocation mechanism for preserving design modularity in object code, that we call *Bind*. Two complementary primitives, *bind* and *remove*, enable our novel invocation mechanism. The *bind* and *remove* primitives are atomic operations that create and destroy the *associations* between program points and a delegate. By association between program points and delegate, we mean that the associated delegate is invoked at these program points. *Bind* is supported in a new aspect-oriented programming model, that we call *Nu*.

We claim and show that, first, it is feasible to realize a programming model that supports *Bind* to preserve design modularity in object code. Second, the new invocation mechanism further improves the conceptual integrity of the programming model by allowing advising and runtime properties of aspect-like constructs to be modeled as sim-

[1]For example, to generate object code that is compliant with the existing virtual machines (VM), such as Java Virtual Machine (JVM) [35] for AspectJ [23] and .NET Framework [38] for Eos [52], to lower the barrier to entry.

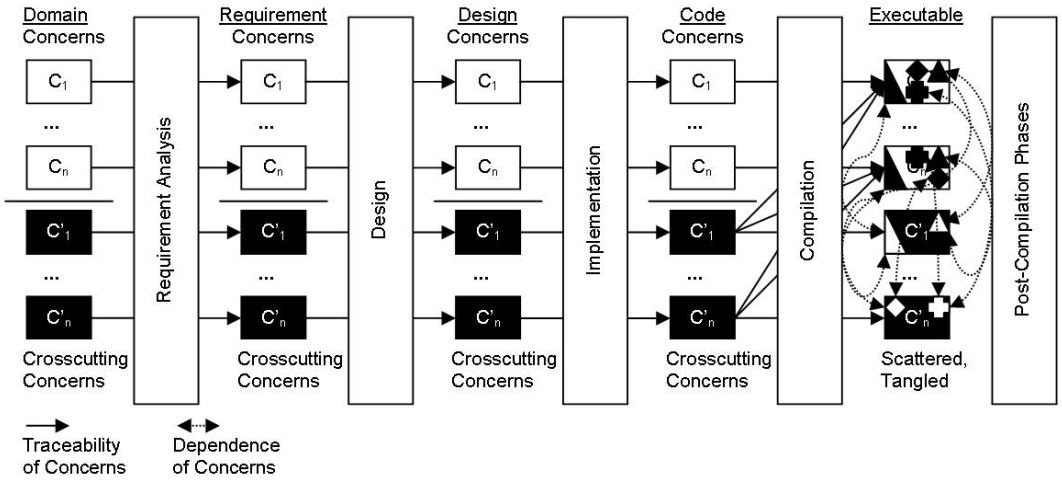

Figure 1: Tracing Concerns through the Life Cycle: In this figure C_1 to C_n and C'_1 to C'_n are the base and the crosscutting concerns respectively. The figure shows that the concerns are separate in the domain, requirement, design and implementation phases. During compilation, to realize the crosscutting behavior $C_1 - C_n$ are modified by inserting calls to and fragments from $C'_1 - C'_n$.

ple combinations of invocation primitives as opposed to new language constructs. Third, it brings a new possibility for structuring aspect-oriented systems, removing the commitment to a single aspect-language model, and expanding the program design space to include arbitrary combinations of language models and advising structures.

We present two alternatives to incorporate this invocation mechanism in existing programming models, first, through an extension of the intermediate language, the language for object code, and second, through an application programming interface (API). Extending the intermediate language with these primitives has significant costs in terms of compiler and virtual machine redesign as well as adoption; however, we argue that the new possibilities offered by this abstraction might potentially outweigh the costs.

To evaluate the feasibility of our ideas and to support our evaluation, we have designed and developed an extension of the .NET Framework [38]. The extension is in the form of an additional layer on top of the existing virtual machine that adds support for the *bind/remove* primitives. The extension is fully backward compatible with existing assemblies.

The rest of this paper is organized as follows. Section 2 describes the problem in detail. Section 3 discusses our approach. Section 4 - 6 discuss emulation of existing language models and advising structures using *bind/remove* primitives and support our claims. Section 7 assesses the nature and potential importance of our results. Section 8 discusses related work. Section 9 concludes.

2. DETAILED PROBLEM STATEMENT

To motivate our approach, we demonstrate in the next subsection the problem with common weaving techniques through a simple example application.

2.1 Scattering and Tangling in Object Code

Figure 2 shows the code for a *HelloWorld* application. We have implemented this application using Eos [18, 48]. Eos is an aspect-oriented extension of C# for Microsoft .NET

```
public class Hello{
  static void Main(string[] arguments){
    System.Console.WriteLine("Hello");
  }
}
```

```
public class Trace{
 pointcut traceMethods(): execution(any any(..));

 static after traceMethods(): trace();

 void trace(){
      System.Console.WriteLine("trace() called");
 }
}
```

Figure 2: A Simple Aspect-Oriented Application

Framework [38] that implements the unified-aspect model proposed by Rajan and Sullivan [52]. Rajan and Sullivan showed that the AspectJ notions of aspect and class can be unified in a new module construct that they called the classpect, and that this new model is significantly simpler and able to accommodate a broader set of requirements for modular solutions to complex integration problems [56].

The binding construct in this model allows modularization of crosscutting concerns. A binding is a mechanism to select a subset of join points[2] in the execution of the program and associate a method to execute at those points. The subset of join points selected by the binding are called subjects of the join point. The method that is associated by the binding to execute at these join points is called the handler of the binding.

Our application has two classpects: *Hello* (shown inside the white box) and *Trace* (shown inside the grey box). The classpect *Hello* declares a method *Main* that prints the string *Hello* on the screen and exits. The classpect *Trace* declares

[2]In the terminology of existing aspect-oriented languages, a *join point* is a *language-defined point* in the execution of a program.

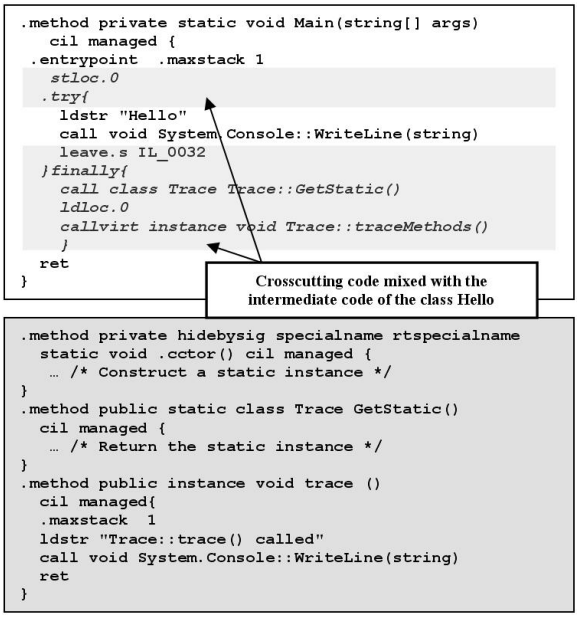

Figure 3: Disassembled HelloWorld

a pointcut *traceMethods* to select all method execution join points in the program and a static binding. The effect of declaring the binding is that the handler method *trace* is invoked at all the subject join points selected by the pointcut *traceMethods* and prints the string *trace() called*. As a result, after the execution of the method *Main* the string *trace() called* is printed.

We compiled this simple application using the Eos compiler [50]. We disassembled the assembly [3] using *ildasm*, the disassembler for .NET Framework. Figure 3 shows the disassembled code. We have used the common intermediate language (CIL) notations to represent the disassembled code. Please note that the weaving techniques for static bindings is similar to that of AspectJ-like languages [23], so the intermediate code shown in Figure 3 is a representative of the current weaving techniques.

Figure shows the disassembled code of *Hello* in the white box and the disassembled code of *Trace* in the grey box. As can be observed, the intermediate code to invoke *Trace* at join points is inserted into the class *Hello* in the method *Main*. As a result, the concern modularized by the classpect *Trace* ends up being scattered and tangled with the *Hello* concern. The separation of *Hello* and *Trace* concerns is thus lost during the compilation phase.

2.2 Effects of Scattering and Tangling

At the minimum, this problem makes efficient incremental compilation and unit testing of AO programs challenging. The best AO compilers available today take significantly more time compared to their object-oriented counterparts for incremental compilation. A recent report on the application of AspectJ [23] to the development of a J2EE web application for Video Monitoring Services of America showed that incremental compilation using the AspectJ compiler usually takes at least 2-3 seconds longer than near instant compilation using a pure Java compiler [30]. It also showed that if an aspect is changed the incremental compilation resorts to full compilation.

The report observed that due to the increase in incremental compilation time, *human attention can wander and it can take time to re-contextualize after the compilation. This problem is particularly pronounced for the full builds, which tempt the programmer to switch to another task entirely (e.g. email, Slashdot headlines).*

The significant increase in incremental compilation time is because when there is a change in a crosscutting concern, the effect ripples to the fragmented locations in the compiled program forcing their re-compilation. For example, let us assume that the source code for the classpect *Trace* changes, so that it now selects all execution join points where the method name begins with a *Set*, for example SetX, SetY, etc. The method *Main* in the *Hello* class is no longer selected by this pointcut for advising.

This change will trigger the incremental compilation of *Trace*. In addition, it will also trigger the compilation of *Hello* to reflect the changes in the pointcut. If the separation of concerns would have been preserved in the intermediate code, it would have been sufficient to just recompile the changed concern, e.g. *Trace*, in the system. The full re-compilation of this simple system is not a huge burden on the program; however, in nontrivial systems the overhead of compilation can be significant enough to disrupt the build-test-debug cycle common in current agile software development processes.

The re-compilation time is affected by two factors. First, increase in the number of crosscutting concerns in a large-scale system. Second, increase in the number of modular concerns that these crosscutting concerns are scattered and tangled with. For a change in a crosscutting concern such as tracing or logging, recompilation of the entire system will be necessary. The system studied by Lesiecki [30] can be classified as a small to medium scale system with just 700 classes and around 70 aspects. In a large-scale system, slowdown in the development process can potentially outweigh the benefits of separation of concerns.

Besides incremental compilation, loss of separation of concerns also makes unit testing of AO programs difficult. The dependence of aspects on other classes and vice versa makes it harder to test them separately. AOSD has shown real benefits in its ability to achieve a separation of some traditionally non-modular concerns. In order to continue receiving these benefits in large-scale systems without impeding the design-build-test cycle common in agile development processes, it is essential to address these issues effectively.

2.3 Learning from the History of Separation of Concern Techniques

This problem is not unique to AOSD; rather it pertains in general to the mechanisms for separation of concerns (SoC) (See Figure 4) [49]. Consider an analogy in the procedural abstraction world. In an instruction set architecture (ISA) that does not support method calls, one could still decompose a program into a set of procedures in the analysis, design and implementation phases. The compiler would then translate these programs into a monolithic set of instructions by in-lining the procedure bodies.

For these programs, benefits of procedural abstraction such as modular reasoning, parallel development, etc., are observed in analysis, design and implementation phases. In

[3] Assembly is a .NET Framework term for an executable.

Separation of Concerns (SoC) Technique	Abstract Invocation mechanism
Procedural Abstraction	Method Call as instruction in ISA
Object-Orientation	Objects and Virtual Method Calls
Aspect-Orientation	An Open Question

Figure 4: Separation of Concern Techniques and Corresponding Abstract Invocation Mechanism

later phases, however, we no longer get the same benefits because there is no clear separation anymore. For example, changes in the source code of a procedure affect all call sites of the procedure, because it is in-lined. Incrementally compiling these procedures was thus harder and more time consuming. Unit testing of such procedures was also a challenge. The support for method call in ISAs, along with the invention and refinement of linking technology has more or less solved these problems for procedural decomposition.

Consider another analogy in the object-oriented world. Like procedural abstraction, object-orientation can be emulated in the analysis, design and implementation phases without the support for objects and dynamic dispatch by translating the OO program into a procedural program that uses methods and structures; however, loosing the traceability of concerns during compilation does affect post-compilation phases. The support for *objects* and *virtual method calls* in the runtime environments improved the intermediate code level design modularity for object-orientation.

For both SoC techniques, emergence of an abstract invocation mechanism at the interface between the language compilers and execution models extended the benefits of SoC techniques to post compilation phases. These mechanisms pushed the decoupling between concerns further down the execution model, abstracting it behind the interface. The nature of the concerns modularized by aspect-oriented techniques dictates that they execute at scattered and tangled points in the execution of the program. The loss of separation at runtime thus seems unavoidable; however, in the rest of the paper we show that a better separation of concerns at the object code level is achievable.

3. THE NU PROGRAMMING MODEL

To preserve the separation of concerns through aspect-oriented compilation, we propose in this work a new programming model that we call *Nu*. We hypothesize that to achieve separation of concern at the object code level it is necessary to provide a precisely specified invocation mechanism that allows representation of crosscutting concerns as modular units in object code. The aspect-oriented constructs in the high-level languages can be expressed in terms of this invocation mechanism without compromising their expressiveness.

Rajan and Sullivan's work on unified aspect model [52] and Eos [18] motivates this hypothesis. The binding construct in the unified aspect model enables modularization of crosscutting concerns. A binding construct uses pointcuts to select join points in the execution of the program and specifies a list of methods to execute at these join points. These bindings allow both type and instance-level advising structures [50] to be represented in AO programs. In principle, if these bindings are provided as an abstraction by the execution model, separation of concerns at the object code level could be enabled for AspectJ and Eos programs. However, the binding abstraction would still be too high-level to be able to model other aspect language models. For instance, runtime-advising structures cannot be modeled. The basic primitive of AspectJ-like language, the advice construct, presents an even higher-level of abstraction. In the rest of this section, we present the new concepts in our programming model.

3.1 Join Point Model

Like most aspect-oriented models, *Nu* adds only one new concept to the underlying language semantics (also called base language [29]) – join points. A join point is defined as a *well-defined point* in the execution of a program. Instead of AspectJ's join point model, we adopt a finer-grained join point model proposed by Endoh et al. [17]. Endoh et al. call the join point model of AspectJ-like languages a *region-in-time* model because a join point in these languages represents duration of an event, such as a call to a method until its termination. They propose a join point model called *point-in-time model* in which a join point represents an instance of an event, such as the beginning of a method call and the termination of a method call [17]. They show that this model is sufficiently expressive to represent common advising scenarios.

In the *point-in-time model* corresponding to AspectJ's *call* join point, there are three join points *call*, *reception*, and *failure*. These three join points eliminate the need for three different types of advice: namely *before*, *after*, and *after throwing* advice. The *before call*, *after call*, and *after throwing call* become equivalent to *call*, *reception*, and *failure* respectively. Similarly, corresponding to AspectJ's *execution* join point, there are three join points *execution*, *return*, and *throw*. For more details about the *point-in-time model*, please see Endoh et al. [17].

Please note that we define a join point as a *well-defined point* in the program, as opposed to a *language-defined point* in the program. This distinction allows us to emulate languages with implicitly defined join point models such as AspectJ, Eos, etc as well as languages and approaches that allow explicitly defined join point models such as SetPoint [4] and annotation based join point models [25] such as JBoss [11]. This requirement puts the burden on the implementer of the *Nu* model to provide a mechanism to define join points to the execution model.

3.2 Bind: An AO Invocation Mechanism

The basis of our approach is an invocation mechanism, that we call *Bind*. *Bind* is enabled by two primitives: *bind* and *remove*. Both these primitives expect two arguments, a *pattern* and a *delegate*. The *pattern* serves to select the subset of the join points in the program. The *delegate* or the *delegate chain* specifies a list of methods that provide the additional code that is to execute at these join points. The *bind* primitive associates the supplied delegate with the join points matched by the pattern. As a result, the delegate

bind	1. Pops top two values from the stack: pattern and delegate 2. Semantics: After this atomic instruction is complete, a set of associations are created between the delegate chain and every join point that matches the pattern.
remove	1. Pops top two values from the stack: pattern and delegate 2. Semantics: After this atomic instruction is complete, if there was a binding between the delegate and the pattern, it is removed.

Figure 5: Extensions to the combined intermediate language (ECIL)

chain is invoked when the program execution reaches the join point. The *remove* primitive eliminates this association.

Please note that at this time we have explicitly decided not to support static crosscutting mechanisms such as inter-type declarations in AspectJ [23]. There are two reasons behind this design decision. First, in most inter-type declarations there is a one-to-one explicit mapping between the classes and the aspects. Therefore, the scattering and tangling of the crosscutting object code is also limited to the class that the aspect affects. Second, inter-type declarations can be emulated using partial classes in C# version 2.0 [15], and mix-ins [8].

3.3 Specificity of bind/remove

Our model does not impose any restrictions on the order and the combination of *bind* and *remove*. In a program execution path, a *bind* may never be followed by a *remove* and vice versa. This allows us to model associations that once created last for the entire execution of the program. As we will show, AspectJ-style [23] advising relationships are modeled using this pattern. Our model also does not require that if a *bind* is followed by a *remove* they provide the reference to the same *pattern* object. This allows us to model the difference operation on the set of associations. A *remove* primitive might eliminate just a subset of the *associations* that was created by an earlier *bind*. The *pattern* object itself is *mutable*.

3.4 ECIL: An Instantiation of Nu

The *bind* and *remove* primitives can be included in the intermediate language (language for object code) as instructions. The intermediate code is essentially an interface between the programming language compiler and the runtime environment. This interface governs the intermediate code that the compiler can generate and the semantics of instructions that it can expect. If included in the intermediate language, these primitives will provide a crosscutting abstraction between the HLL compiler implementation and the runtime environment. This interface will then abstract the realization of the crosscutting behavior at run-time from the HLL compiler implementations. The interface will also govern the semantics of the crosscutting primitives that the compiler implementation can expect for code generation purposes.

Figure 5 shows these primitives as an extension of the common intermediate language (CIL), the intermediate language for the .NET Framework. We call this extension ECIL. The *bind* instruction expects a reference to a *delegate* and the reference to a *pattern* to select join points as the top two items on the stack. The pattern is equivalent to a pointcut expression. When the instruction is complete an *association* between the join points selected by the *pattern* and the *delegate* is created. The effect is that the delegate is invoked at all join points selected by the pattern. The *remove* instruction also takes the reference to a delegate and a pattern and eliminates the association.

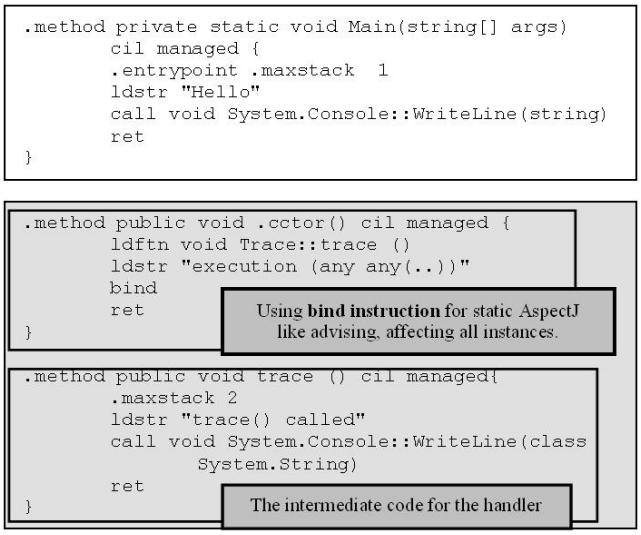

Figure 6: ECIL version of HelloWorld

3.5 Revisiting HelloWorld

To illustrate, let us revisit our *HelloWorld* example. The intermediate code for the application in the ECIL is shown in Figure 6. Like before, the disassembly of classpect *Hello* is shown in the white box and the disassembly of classpect *Trace* is shown in the grey box.

Instead of explicit callbacks in the intermediate code for *Hello*, *bind* and *remove* instructions are generated in the intermediate code for *Trace*. Note that we are translating a static binding that affects all instances of *Hello*. To model the semantics of static binding, a *bind* instruction is inserted in the static constructor of *Trace*. The constructor pushes a delegate to the method *trace* on to the stack followed by the string *execution (any.any(..))*. The *bind* instruction follows these two push instructions. As a result, when the type *Trace* is initialized, the handler *trace* is associated to execute at the selected subject join points.

At this time, we are using a string to represent the pattern. One might argue that this design decision gives away the type safety. We agree that this might be a problem; however, we would like to emphasize that ECIL code will not be hand-written. A compiler will translate the code from a high-level language to ECIL. This high-level language can include a sub-language such as the pointcut sublanguage of AspectJ to express the patterns. The compiler would then check the syntax and the semantics of the patterns and then generate a string corresponding to the checked pattern. Nevertheless, for this and other reasons such as verification of pointcut description by the runtime and the runtime's efficiency of interpreting the pointcut, we provide a language

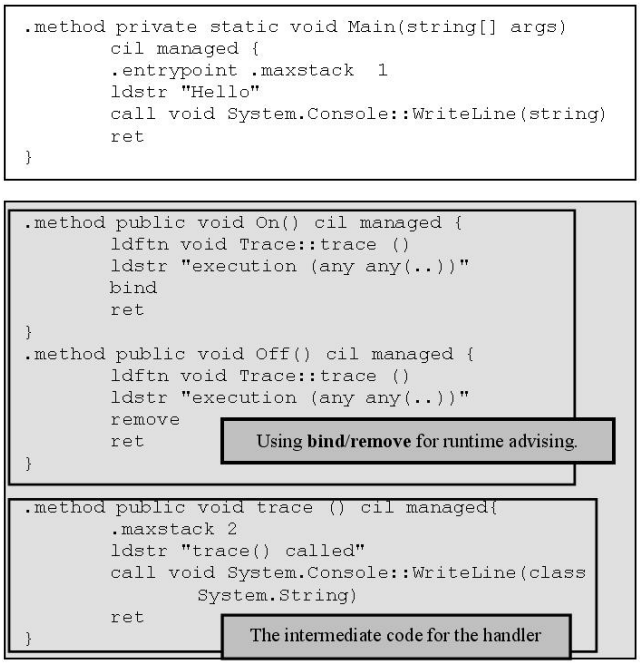

Figure 7: Implementing Runtime Advising in ECIL

for expressing the pattern in our alternative instantiation of *Nu* that we will discuss later in this paper.

Our example demonstrates that the separation of concerns is preserved for modules represented in ECIL. The code for *Hello* is free of the callbacks to the *trace* method in the *Trace* classpect. As a result a change in *Hello*, which is not a crosscutting concern, will only affect the intermediate code representation of the *Hello* module. Similarly, a change in *Trace*, which is a crosscutting concern, will only affect the intermediate code representation of the trace module. The changes are thus traceable to a limited number of modules at the intermediate code level, resulting in an improved incremental compilation time compared to existing approaches.

3.6 Emulating Runtime Advising

The example we presented above demonstrates static advising. The *bind/remove* primitives can also be used for runtime advising (See Figure 7). The figure shows a variation of our *HelloWorld* application. Now we want to enable and disable tracing at runtime. To do that, the modified classpect *Trace* provides two methods, *On* and *Off*. The ECIL representation of method *On* consists of instructions to push the delegate to the handler and the pattern on the stack followed by the *bind* instruction (similar to the static constructor implementation in the static tracing example). The method *Off* also consists of instructions to push the delegate and the pattern followed by the *remove* instruction. The semantics of *bind* and *remove* ensure that calling *On* activates the tracing and calling *Off* deactivates it.

3.7 Bind API: Another Instantiation of Nu

An alternative instantiation of *Nu* is to provide the invocation mechanism *Bind* as an application-programming interface (API). Our realization of the *bind/remove* primitives as an API is shown in the Figure 8. The semantics of *Bind/Remove* methods of the API are similar to the *bind/remove* instructions of the *ECIL*. The only difference is that instead of using strings to represent the pattern, this API provides a library of patterns that implement the *IPointcut* interface. The *IPointcut* interface is shown in the Figure 9. This interface obligates the implementer to provide a method to match an object of type *IJoinpoint*. An object of type *IJoinpoint* represents a join point at runtime.

```
1  void Bind(IPointcut pcut, SimpleDelegate delegate);
2  void Bind(IPointcut pcut, JPDelegate delegate);

3  Semantics: After this atomic method returns,
4  a set of associations are created between
5  the delegate chain and every join point that
6  matches the pattern.

7  void Remove(IPointcut pcut, SimpleDelegate delegate);
8  void Remove(IPointcut pcut, JPDelegate delegate);

9  Semantics: After this atomic method returns, all
10 associations between the delegate chain and every
11 join point that matches the pattern are removed.
```

Figure 8: The Bind API

```
1  public interface IPointcut {
2      bool Match(IJoinpoint thisJP);
3  }
```

Figure 9: The IPointcut Interface

At this time, we have implemented two variations of delegate chains: *SimpleDelegate* and *JPDelegate*. The delegatees in the delegate chain of type *SimpleDelegate* cannot receive any arguments. For some use cases, however, the delegatees might need some reflective information about the join point. We provide the delegate chain type *JPDelegate* for this purpose. The delegatees in the delegate chain of type *JPDelegate* may receive an argument of type *IJoinpoint*. The reflective information can be marshaled from this argument. This is a limitation of the current model that sacrifices static type checking on delegatees arguments. We plan to fix this problem in future versions of the *Bind* API. The recent support for generics in both Java and C# appears to be a promising direction to address this issue.

3.8 Emulating Instance-Level Advising

The examples we presented in the context of *ECIL* demonstrate emulation of static and runtime advising structures using our model. The *bind/remove* primitives can also be used for instance-level advising (See Figure 10). Rajan and Sullivan define instance-level advising as the ability to differentiate between two instances of a class, while advising, and to advise them differently if needed [51]. The figure shows a variation of our *HelloWorld* application. To demonstrate, we have re-factored the functionality to print the string `hello` to a new class *Hello*. The main program now creates three instances of the class *Hello*: *h1*, *h2*, and *h3*. We now want to add the functionality of printing the string `world` after the string `hello` only when the method *Say* is called on the instance *h2*.

To achieve that we add a new class *World* (lines 14-26). The class defines a method *Add* (lines 15-22) for the main program to specify which *Hello* instance is to be advised.

```
1  class HelloWorld {
2    ...
3    static void Main(string[] args) {
4      /* Create h1, h2, h3, 3 instances of Hello*/
5      World.Add(h2); /* Add saying world only to h2 */
6      ...
7    }
8  }
9  class Hello {
10  public void Say() {
11    System.Console.WriteLine("Hello");
12  }
13 }
14 class World {
15  static void Add (Hello h) {
16   Bind(
17      And(
18   new Return(
19          new MethodPattern(Public,"Hello.Say")),
20        new This(h)),
21      new SimpleDelegate(World.Say));
22  }
23  public static void Say() {
24    System.Console.WriteLine("World");
25  }
26 }
```

Figure 10: Implementing Instance-Level Advising

This method calls the *bind* primitive (lines 16-21) supplying it a *pattern* and a *delegate*. The delegate (line 21) is a *SimpleDelegate* containing only one method *Say* (lines 23-25) of the class *World*. The pointcut provided as pattern (lines 17-20) is a composition of two sub-pointcuts using the binary *And* operator (line 17). The *And* operator is similar to AspectJ's && operator, except that our operator is a first-class construct. A join point is matched by the result of the *And* operator, if and only if it is matched by both operands supplied as arguments. Our pattern library also provides *Or* and *Not* operators.

The two operands to the *And* operator are a *Return* pointcut (lines 18-19) and a *This* pointcut (line 20). The *Return* pointcut matches the return join point of the method *Say* of the class *Hello*. The *This* pointcut (line 20) matches any join point where the executing instance is *h*. The combination of these two pointcuts using *And* matches join points that are *return* join points where the executing instance is *h*. This is precisely the definition of selectively advising instance *h*. Our *Bind* API was thus able to emulate *instance-level advising* as well. In addition, the compiled code for the class *HelloWorld*, *Hello* and *World* remain separate. The API abstracts the scattering and tangling. Thus, our *Bind* API also preserves the design modularity at the object code level.

3.9 Summary

In this section, we presented a new programming model called *Nu* for preserving design modularity at object code. The separation of concerns at the object code level is preserved by abstracting scattering and tangling behind a new aspect-oriented invocation mechanism called *Bind*. This invocation mechanism is enabled by two primitives *bind* and *remove*. We showed that the unified model implemented by Eos can be supported using *Bind*. We also showed that *static*, *runtime* as well as *instance-level* advising can be expressed in terms of *Bind*, supporting our claim that our invocation mechanism allows advising and runtime properties of aspect-like constructs to be modeled as simple combination of invocation primitives as opposed to new language constructs.

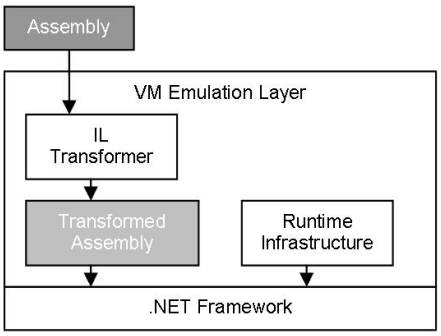

Figure 11: Current Nu Implementation

4. EVALUATION

In Section 1, we made three claims about our approach. First, it is feasible to realize a programming model that supports *Bind* to preserve design modularity in object code. Second, the new invocation mechanism further improves the conceptual integrity of the aspect-oriented programming models by allowing advising and runtime properties of aspect-like constructs to be modeled as simple combinations of invocation primitives, as opposed to new language constructs. Third, it brings a new possibility for structuring aspect-oriented systems, removing the commitment to a single aspect-language model, and expanding the program design space to include arbitrary combinations of language models and advising structures.

We have already supported our second claim in Section 3, by showing how a variety of advising structures can be implemented using intuitive patterns that uses our proposed invocation mechanism. To validate the first claim, we describe a prototype implementation of the proposed invocation mechanism in Section 5. To confirm the third claim, the implementation of common advising structures in prevalent aspect language models using the proposed invocation mechanism is demonstrated in Section 6.

5. IMPLEMENTATION

To support our first claim that it is feasible to build support for *Nu* in a runtime environment, we implemented the *Bind* API as an additional layer on top of the .NET Framework [38]. This layer can be added to the shared source common language infrastructure [2], an open implementation of the .NET Framework, however, for the purpose of this evaluation we will consider it separately.

Figure 11 shows the current implementation of the *Nu* model. The two key components of the layer are the IL transformer and the runtime infrastructure. The IL transformer can be considered as an extension of the just-in-time compiler to support *Bind*. It prepares the input assembly to execute inside the existing .NET framework. The runtime infrastructure provides the supporting functionality.

5.1 IL Transformer

As of this writing, the IL transformer is implemented as a post processor for the .NET Framework. It takes a .NET assembly and instruments all join points in the assembly

to construct an object of type *Nu.Runtime.Joinpoint*. This object represents the reflective information about the join point. A call to the *Join Point Dispatcher* is also inserted at all join points. Instrumenting all join points is inefficient compared to the techniques used by AspectJ and Eos compilers. These compilers only instrument the join points that are potentially of interest to bindings. In the future, we will implement optimizations to reduce this overhead.

The IL transformer uses the *program executable file reader/writer application programming interface* (PERWAPI) [21] to manipulate .NET assemblies. PERWAPI provides an API to read and construct assemblies on .NET Framework [38]. This API is used to instrument an input assembly, to include a trigger mechanism at every joint point in the program. An alternative implementation can also use the *System.Reflection* API of the .NET Framework [38] to read and write assemblies on the fly.

5.2 Runtime Infrastructure

Figure 12 shows the runtime infrastructure that implements the core of the *Bind* API. In Nu, all join points invoke a *join point dispatcher* as shown in Figure 12. A *join point dispatcher* is similar to event dispatcher used in the modeling of implicit invocation systems [14]. There is only one instance of the dispatcher in an application's address space. The figure shows join points J_0 .. J_n being dispatched.

The current implementation of the dispatcher also stores the *patterns* and corresponding *delegates*. On a *bind* call, which is equivalent to the *bind* instruction execution in ECIL, the argument *pattern* and *delegate* are stored. On a *remove* call, if the *pattern* and *delegate* pair are present they are removed from the store. On a join point dispatch, the dispatcher attempts to match the join point with the stored patterns. On finding a match, the corresponding delegate chain is invoked.

5.3 Summary

The current implementation of our model is preliminary and somewhat inefficient; however, it still suffices to show the feasibility of supporting *Bind* as an invocation mechanism. There are several promising directions to address the efficiency issues such as set-based storage of (*pattern,delegate*) so that a match becomes equivalent to set-membership that is faster to compute. There are also opportunities to learn from other systems. For example, in the context of implicit invocation systems, several algorithms exist for efficient and fast matching of events to subscribers. Pereira et al. [44] proposed a semi-structured event model and a predicate matching algorithm in the context of their Publish/Subscribe system *Le Subscribe*. Campailla et al. [12] propose an efficient event-matching algorithm based on binary decision diagram. A combination of these approaches can also be used for efficient matching of join points.

6. EMULATING OTHER AOP MODELS

We claimed in Section 1 that *Nu* brings new possibility for structuring aspect-oriented systems, removing the commitment to a single aspect-language model, and expanding the program design space to include arbitrary combinations of language models and advising structures. In this section, we present realizations of prevalent aspect-oriented language models using *Bind* to support that claim. The aspect-language models that we discuss in this section include HyperJ [41], Composition Filters [3], Adaptive Programming [34], and AspectJ [23]. We have already shown in Section 3 that the unified aspect model of Eos [52] can be realized using the *Bind* API. To make this work self-contained, we present a brief overview of these AOP models.

6.1 Emulating the HyperJ Model

HyperJ [57] supports multi-dimensional separation of concerns through the use of *hyperslices* [58] and *hypermodules*. [41] A *hyperslice* represents a single concern in a system. It may be composed of multiple classes and span multiple packages. The concerns are identified in a concerns file which maps portions of a system to the concerns those portions implement. Multiple concerns may then be composed together to form a *hypermodule*. A *hypermodule* usually consists of multiple *hyperslices* and states how HyperJ should compose the various concerns implemented by those hyperslices to create the final system. Units with the same name can be merged, override each other, or have no correspondence at all. A *hypermodule* defines the default method of composition and may then specify exceptions for units by using *Merge*, *Override*, *NoMerge*, etc. A *hypermodule* may also state that a method x() will *bracket after* or *bracket before* method y(), meaning that x() will be executed after or before the method y() executes, respectively.

6.1.1 HelloWorld in HyperJ

Figure 13 shows an example of a HyperJ program. This program consists of three hyperslices: *Feature.Main*, *Feature.Hello*, and *Feature.World*. The concerns in these hyperslices are defined based on the operations *Main*, *Hello*, and *World*. These three hyperslices are then composed in the *HelloWorldHM* hypermodule. This hypermodule states that there is a specific order such that the *Feature.Hello* hyperslice should occur before the *Feature.World* hyperslice. This hypermodule also states that the methods *Feature.Hello.Hello()* and *Feature.World.World()* should be bracketed *after* the method matching the pattern *Main* in any package.

If we assume that the semantics of the *Feature.Hello* concern states that the string *Hello* will be printed on the console and the *Feature.World* concern states that the string *World* will be printed on the console, it should be clear that the expected behavior of this HyperJ program would be to print the string *Hello* followed by the string *World* on the console.

6.1.2 HyperJ-Style HelloWorld using the Bind API

Figure 14 shows what the example program might look like if compiled from HyperJ to the *Bind* API. The classes *HelloWorld*, *Hello*, and *World* represent the three hyperslices *Feature.Main*, *Feature.Hello*, and *Feature.World* respectively. The class *HelloWorldHM* represents the hypermodule *HelloWorld*. The class makes use of the *Bind* API to bracket the methods *Hello.Hello()* and *World.World()* after the method *Main*. The ordering between the *Feature.Hello* and *Feature.World* hyperslices is maintained by the semantics of the *Bind* API.

This simple example demonstrates how one could simulate constructs in the HyperJ model using the *Bind* API. The semantics of the *Bind* instruction guarantees that after it is called in the example, for all join points matching the *return* of a *static* method named *Main*, the del-

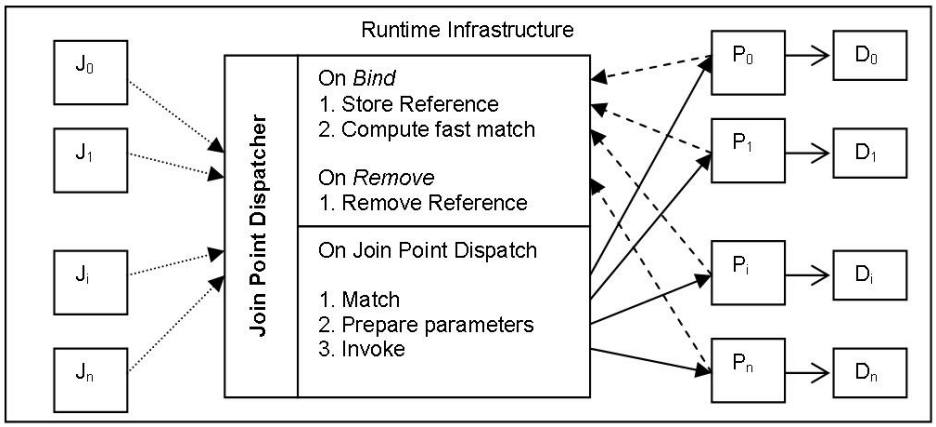

Figure 12: The Supporting Runtime Infrastructure

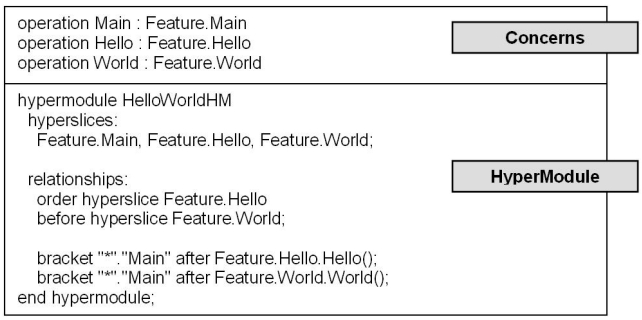

Figure 13: HyperJ version of HelloWorld

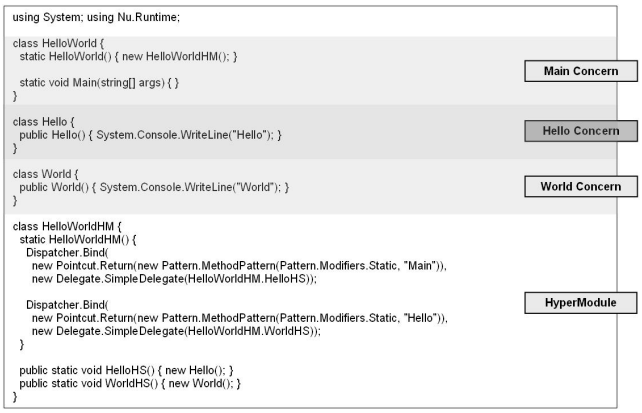

Figure 14: HelloWorld Using Bind API

egatee *HelloWorldHM.HelloHS* is invoked and for all join points matching the constructor of *Hello*, the delegatee *HelloWorldHM.WorldHS* will be invoked. These in turn create new objects of type *Hello* and *World* respectively. The *Hello* and *World* classes fulfill our semantic requirements of the *Feature.Hello* and *Feature.World* concerns by outputting the strings *Hello* and *World* respectively to the console.

HyperJ has three default composition strategies: *mergeByName*, *nonCorrespondingMerge*, and *overrideByName*. *nonCorrespondingMerge* is obviously supported. *overrideByName* (and of course *Override*) could be simulated by providing empty methods for each class. A return point-cut on those methods would then use the appropriate method (the overriding method) as advice. *mergeByName* (and of course *Merge*) could be simulated in a similar manner. Instead of the advice only calling the overriding method however, it would call all matching methods.

HyperJ *summary functions* could be supported by the addition of the *skip* construct defined by Endoh et al. [17] The results of each merged operation could be stored in an array and sent to the summary function, who's result would be used as the final return value. *Equate* and *Match* simply set up join point aliases and could be supported. *Rename* operates on the composed names and also could be supported. *Order* and *Bracket* were shown in the example above and *NoMerge* is obviously supported.

6.2 Emulating the Composition Filter Model

The composition filters (CF) model [3] extends the object-oriented model by introducing a new layer called *interface* between an object and its clients. This layer consists of input and/or output filters as well as internal and possibly external objects. The behavior of an object is composed with the behavior of these objects. The composition is performed by manipulating the messages received and sent by the object using input and output filters respectively.

This model creates a first-class representation of a message for manipulation by a set of ordered filters until it is either discarded or dispatched. A filter may define the actual semantics for accepting or rejecting a message. A *filter* specification follows:

<name>:<filter-type>={filter-elem, filter-elem, ...}

A *filter element* is of the form:

$< condition >=> [< match-target > . < match-sel >] < target > . < sel >$

or $< condition >=>< match-target > . < match-sel >$

Note that the model evaluates the filter elements from left to right. The *condition* is a boolean value determined at run-time. *match-target* and *match-sel* are used to match a message against the specified pattern. If *target* and *sel* are specified, they are used to re-dispatch the message as a parameter to the event(s) specified by *target* and *sel*.

```
1  class Log interface
2    conditions
3      Debug;
4    methods
5      Exception(String) returns Nil;
6      Warning(String) returns Nil;
7      DebugOn returns Nil;
8    inputfilters
9      disp: Dispatch = {Debug=>inner.*,
10             True=>inner.Exception,inner.DebugOn};
11     queue: Wait = {Debug=>inner.Warning};
12 end;
```

Figure 15: Composition Filters Logger

6.2.1 Logger in CF Model

Figure 15 shows an example of a Composition Filter program. For this example we are only concerned with the classe's *interface*. The *interface* defines one condition on line 3 named *Debug*. The *interface* also defines three methods on lines 5-7: *Exception*, *Warning*, and *DebugOn*. On lines 9-11, the *interface* defines two input filters *disp* and *queue*.

The *disp* input filter is of type *Dispatch*. This filter type states that upon acceptance of a message, the message will be dispatched to the target. Upon rejection of a message, the filter will forward the message on to the next filter. If there are no more filters then an exception will be thrown. [3] The filter has two filter events. The first filter event will match when the condition *Debug* is true on any messages from the Log implementation. The second filter event will match on the messages *Exception* and *DebugOn* (since the condition is always true).

The *queue* input filter is of type *Wait*. This filter type states that upon acceptance of a message, the message will be forwarded to the next filter (if one is available). Upon rejection of a message, the filter will queue the message and try to accept it at a later time. [3] The filter has one filter event that will match when the condition *Debug* is true on the message *Warning*.

The purpose of this module is to provide selective logging. If the system is in a debug state then clearly all messages should be logged. However, if the system is in a production state then only the most extreme messages (such as exceptions) need to be logged. This is accomplished by the first filter always accepting *Exception* and only accepting *Warning* when *Debug* is true. The second filter stops the exception that would be thrown if the *Warning* message was sent while *Debug* is false.

6.2.2 CF-Style Logger using the Bind API

Figures 16 and 17 show how the example program in Figure 15 might look if compiled from Composition Filters to the *Bind* API. Please note that these figures only show the most important portions of the program.

Examining Figure 16, lines 10-16 show the creation of each filter and their respective filter events. Each filter event's condition gets a method generated (lines 19-20) and a delegate (lines 24-25). The module's condition (*Debug*) is defined on line 22. The base *Filter* class (which is abstract) is defined on lines 44-64. It does not define the *Accept* or *Reject* methods. The class makes use of the *Bind* API to advise the execution of the *Send* method of each *Message*. It will then try to accept or reject the message based on the

```
1  class Logger {
2    static void Main(string[] args) {
3      Log logger = new Log("console");

4      logger.Warning("Something may be wrong.");
5      logger.Exception("Something most definitely is wrong.");
6  } }

7  class Log {
8    private bool isDebug = false; private Filter disp, queue;

9    public Log(string path) {
10     // create the filters in reverse order
11     queue = new WaitDispatchFilter();
12     queue.AddFilterElem(new FilterElem(new Check(disp_Check_1),
13       "Log", new string[] {"Warning"}));

14     disp = new DispatchFilter(queue);
15     ...

16     disp.Enable(); // *first* filter gets a call to enable
17   }

18   // condition checks
19   private bool disp_Check_1() { return Debug(); }
20   private bool disp_Check_2() { return true; }

21   // conditions
22   public bool Debug() { return isDebug; }

23   // delegate declarations
24   private delegate void logDel1(string s);
25   private delegate void logDel2();

26   // body definitions
27   private void ExceptionBody(string s) { ... }
28   private void WarningBody(string s) { ... }
29   private void DebugOnBody() { ... }

30   // message invocations
31   public void Exception(string s) {
32     Message m = new Message("Log", "Exception",
33       new logDel1(ExceptionBody), new object[] {s});
34     m.Send();
35   }
36   public void Warning(string s) { ... }
37   public void DebugOn() { ... }
38 }

39 public class Message {
40   ...

41   public void Send() { }
42   public object Dispatch() { return body.DynamicInvoke(args); }
43 }

44 abstract public class Filter : IFilter {
45   ...

46   public void Enable() {
47     Dispatcher.Bind(Pointcut.Execution(
48       Pattern.MethodPattern(Pattern.Modifiers.Any, "Send")),
49       Delegate.JPDelegate(TrackMessage));
50   }

51   public void TrackMessage(IJoinpoint thisJP) {
52     Match((Message)thisJP.This);
53   }

54   public void Match(Message m) {
55     foreach (FilterElement element in filterElements)
56       if (element.condition() && m.Target.Equals(element.Target))
57         foreach (string s in element.Selections)
58           if (s.Equals(m.Selection)) {
59             Accept(m);
60             return;
61           }

62     Reject(m);
63   }
64 }
```

Figure 16: Logger using Bind API

```
1  public class DispatchFilter : Filter {
2    ...

3    public override void Accept(Message m) {
4      m.Dispatch();
5    }

6    public override void Reject(Message m) {
7      if (nextFilter == null) throw new FilterException(...);

8      nextFilter.Match(m);
9    }
10 }
```

Figure 17: The Dispatch Filter

filter elements it contains. If a message is dispatched, the *Dispatch* method (line 42) is called which invokes the body of the message. Note that the *Send* method does not actually do anything and is used for defining a join point for the message.

Figure 17 shows how to implement the *Dispatch* filter. The semantics of the filter state to dispatch a message if it is accepted (line 4). If the message is rejected, pass it on to the next filter (line 8). If there are no more filters then throw an exception (line 7).

Support for the three target types (*inner*, *internal object*, and *external object*) is available using the *Bind* API. Support for the *inner* target was shown in the example. Support for *internal object* and *external object* targets is straightforward using standard object oriented techniques such as sub-classing and composition, respectively.

Although only some of the standard filter types (*Dispatch*, *Error*, *Wait*, and *Meta*) are currently supported, it is straight-forward to add new filter types since the filters define the semantics of accept and reject.

Superimposition is not directly supported at this time, however this concept could be simulated in the future using the *Bind* API. The *selectors* could be simulated by the addition of the *skip* construct defined by Endoh et al. [17] The advice of the aspects would then create the appropriate filters and fire the appropriate message. Only if the message is dispatched would the advice call *proceed*.

6.3 Emulating Adaptive Programming

The adaptive programming (AP) model is an extension to object-oriented model. The main idea behind the AP model is to *separate the program text and the class structure* [43]. The adaptive program is a collection of propagation patterns, which encapsulates the behavior of a concern. Each propagation pattern is composed of traversal strategies and code wrappers. The former selects the objects that will be affected by the concern according to their classes, thus the AP model defines the join points as instance of the class. The objects are selected by traversing the class graph, which is a graph representing the relations (is-a, has-a relationships) between the different classes of the system. As for the latter (i.e. code wrappers), they associate actions to the selected objects [42] (similar to advices in AspectJ).

6.3.1 An Example in Adaptive Programming

An example adaptive program is shown in Figure 18 [32]. This example is written in Java using the DJ library, a library in the Demeter project [39]. The purpose of the code is to sum the values of all the *Salary* objects reachable by

```
1  class Company {

2   // class structure
3   static ClassGraph cg = new ClassGraph();
4   Double sumSalaries() {

5     // traversal strategy
6     String s = "from Company to Salary";
7     Visitor v = new Visitor() {

8       // adaptive visitor
9       private double sum;
10      public void start() { sum = 0.0 };
11      public void before(Salary host)
12          { sum += host.getValue(); }
13      public Object getReturnValue()
14         { return new Double(sum); }
15    };
16    return (Double) cg.traverse(this, s, v);
17  }
18  // ... rest of Company definition ...
19 }
```

Figure 18: sumSalaries Adaptive Method [32]

```
1  class AdaptiveSalarySum {
2     static Double sum;
3     static bool FromEncountered = false;
4     static AdaptiveSalarySum() {
5         Dispatcher.Bind(
6             new And(
7               new Pointcut.Execution(
8                 new Pattern.MethodPattern(
9                     Pattern.Modifiers.Public, "Traverse")),
10              new Pointcut.Args(
11                  System.Type.GetType("Payroll.Company"))),
12            new Delegate.SimpleDelegate(AdaptiveSalarySum.from));

13            Dispatcher.Bind(
14              new And(
15                new Pointcut.Execution(
16                  new Pattern.MethodPattern(
17                      Pattern.Modifiers.Public, "Traverse")),
18                new Pointcut.Args(
19                    System.Type.GetType("Payroll.Salary"))),
20              new Delegate.JPDelegate(AdaptiveSalarySum.to));
21    }
22    public static void from(){
23       FromEncountered = true;
24       sum = 0.0;
25    }
26     public static void to(IJoinpoint thisJP) {
27        if (FromEncountered){
28          Payroll.Salary sal = (Payroll.Salary)thisJP.Args[0];
29          sum += sal.getValue();
30        }
31     }
32 }
```

Figure 19: Emulating Adaptive Programming

```
1 aspect Tracing {
2   pointcut tracePointcut():
3     execution(* *.SayHello());
4   after(): tracePointcut() {
5       System.out.println("World");
6     }
7 }
```

Figure 20: Tracing Aspect in AspectJ

has-a relationships from a *Company* object. In the Demeter project, a propagation pattern is called an adaptive method, and the code wrapper is implemented using the Visitor pattern, and thus called an adaptive visitor.

In this example, the *ClassGraph* constructor (line 3) constructs the class structure. The traversing of the class graph (line 16) starts from *this* object, traverses the path *from Company to Salary* and executes the applicable visitor methods along the way. The adaptive visitor (lines 7-15) initializes the sum upon starting, adds the value to the sum when reaching each *Salary* object, and returns the value upon finishing.

6.3.2 Emulating Adaptive Programming in Bind API

Figure 19 shows the emulation of the adaptive programming example shown in Figure 18 using the *Bind* API. Like the original example, the classgraph is created and traversed. The first *Bind* instruction (lines 5-12) checks whether the *Traverse* method is executed and that the first parameter is of type *Payroll.Company*. If so, the delegate method *from* (lines 22-25) is invoked. The latter sets the flag *FromEncountered* to true, thus indicating the traversal was started from the *Company* node in the class graph and initializes the *sum*. The second *Bind* instruction (lines 13-20) checks whether the *Traverse* method is executed and that the first parameter is of type *Payroll.Salary*. If so, the delegate method *to* (lines 26-31) is called. This method retrieves the first parameter of the join point (which is an instance of the salary) and adds the value of it to the sum only when *FromEncountered* is true, thus making sure that the class graph is traversed starting from *Company* before reaching the instance of *Salary*.

The experiment above showed that to a certain extent, adaptive programming structures can be expressing using the *Bind* API. There is still a need for building and traversing class graphs, however, these functionalities can be provided as a re-usable component.

6.4 Emulating the AspectJ Model

AspectJ [23] is an aspect oriented extension to Java [20]. It is a representative language in the broader class of Pointcut-Advice-based AO languages [36]. Other languages in this class include AspectC++ [55], AspectR [10], AspectWerkz [7], AspectS [22], Caesar [37], etc. Please note that these languages use a *region-in-time* join point model as opposed to a *point-in-time* model.

A simple aspect in AspectJ is shown in Figure 20. The aspect uses the pointcut (lines 2-3) to select the execution of any method named *SayHello()* during the execution of the program. An advice (see lines 4-6) executes at all join points selected by *tracePointcut* and writes the string `World` on the screen.

In *Nu*, this programming style can be emulated as shown

```
1 class Tracing {
2  static IPointcut tracePointcut =
3      new Return(new MethodPattern(Any,"SayHello"))
4  static Tracing () {
5   aspectInstance = new Tracing();
6   Dispatcher.Bind(
7      tracePointcut,
8      new SimpleDelegate(aspectInstance.SayWorld));
9  }
10 public static Tracing aspectInstance;
11 ...
12 public void SayWorld() {
13   System.Console.WriteLine("World");
14 }
15 }
```

Figure 21: Emulating AspectJ-Style AOP

in Figure 21. In this figure, the aspect *Tracing* is modeled as a class. This aspect is a singleton aspect. Under the hood, the AspectJ compiler implements the *singleton design pattern* [19, pp. 127] and makes two methods *hasAspect* and *aspectOf* available to programmers. Every advice executes on the singleton instance, *aspectInstance*. We emulate the programming style using the *Bind* API. We provide a method *SayWorld* to write the string `World` on the screen and a singleton instance *aspectInstance* on which this method is called. The static constructor of the *Tracing* class initializes this instance and calls the *bind* primitive to create an association between all join points selected by the *tracePointcut* and the delegate. For this emulation, the knowledge of the *singleton design pattern* and the *bind* primitive is enough. AspectJ's language model is by far the most expressive model of aspect-oriented languages. Our current implementation cannot emulate the entire language because some constructs like *cflow* and *within* and join points like *exception handler* are not currently supported. Nevertheless, the current experiments suggest that at least the core constructs, namely advice, pointcut, and aspect, in the AspectJ language model can be emulated using the *Bind* API.

6.5 Summary of Results

In this section, we showed how different language models can be emulated using the *Bind* API. A part of the AspectJ model was emulated using the Singleton design pattern [19, pp. 127]. The adaptive programming was emulated using the propagation pattern [33]. A subset of the features of the composition filters model and the HyperJ model were also emulated similarly.

Some emulation examples were not so straightforward. In particular, emulating the composition filter and adaptive programming models required developing code for *classgraphs* and *filters*. However, most of the code we developed is reusable. For example, the code to construct the *classgraph* is generic and can be used to write other programs in the adaptive programming style using the *Bind* API. Similarly, the abstract class *Filter*, *Message*, etc is also reusable. In future, we will provide libraries to emulate these prevalent programming models using *Bind*.

The key property of this result is that emulating one programming model and utilizing design structures that comes with it does not restrict the designer from using another programming model. For example, using adaptive programming to separate the salary summation concern doesn't rule out the possibility of using perhaps AspectJ-style design

structures for method execution tracing, or HyperJ style composition for composing *Employee's* functionality with a *Person's* functionality to construct a more humane payroll system that accounts for gender based necessities while computing the net salary. Due to pragmatic reasons such as commitment to a set of compatible tool support, such design structures have been difficult to realize so far [59].

With new possibilities come new problems. Being able to reason about a system that uses two or more AOP models at once is possibly more complicated then just the sum of the efforts required to reason about them individually. The integration between the language models also need to be accounted for. Fortunately, research results have started to emerge to address these issues. For example, recently Kojarski and Lorenz [27] proposed a semantic framework to compose multiple domain-specific AO extensions together. To a certain extent, their results apply to our programming model as well.

7. DISCUSSION

It is perhaps very common in aspect-oriented programming research literature to provide language extensions to support new properties of aspect-like constructs e.g. instantiation, instance-level advising, runtime advising, etc. For example, Rajan and Sullivan [50] provided modifiers *instance-level* to model instance-level weaving, later Sakurai et al. provided similar extensions using *target* pointcuts [54], AspectJ [23] has added numerous extensions such as *perthis*, *pertarget*, etc, to model similar properties. Extending the language every time to provide new constructs to model these properties works, however, it increases the conceptual burden on the programmer. In this work, we viewed these properties as patterns that provide effective representation of certain design structures in terms of a more fundamental primitive. This view decreased the conceptual burden, requiring programmers to learn only one new primitive well at first and learning other available patterns on a need basis.

The notion of conceptual integrity in design provides a basis for these observations. Brooks wrote:

> ...that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, then to have one that contains many good but independent and uncoordinated ideas. ... Simplicity and straightforwardness proceed from conceptual integrity. Every part must reflect the same philosophies and the same balancing of desiderata. Every part must even use the same techniques in syntax and analogous notions in semantics. Ease of use, then, dictates unity of design, conceptual integrity." [9, pp. 42-44].

By providing a simple set of primitives, and allowing other higher-level constructs to be derived from it using similar techniques, our programming model improves the conceptual unity of aspect-oriented programming models.

We finish this section by addressing the rationale to maintain compliance with existing virtual machines. The goal was adoptability. Allowing users to be able to use aspect-oriented techniques and not having to throw away their virtual machine implementations was a valid reason for early adopters. Aspect-oriented techniques have gained significant visibility since then, and shown potential [53]. Small to medium scale software systems are built using these techniques. The cost of not preserving design modularity in object code is not yet a commonly visible impediment because the scale is small, however, traces are becoming evident [30].

The benefits of aspect-oriented modularity are attractive at a large scale; in fact, the majority of the benefits only become apparent during a large-scale use such as IBM WebSphere [13]. However, large-scale usage does come with unique performance requirements such as *design-build-test cycle time*, *full build time*, etc. The best AO compilers available today have pushed the performance limits and delivered significant improvements compared to early versions, however, there is only so far they can go without addressing the underlying fundamental problem. Once these issues become apparent to the large-scale adopters, tough decisions on the tradeoff of separation of concern vs. performance will have to be made. It is too risky to leave such decisions in possibly incapable hands. Perhaps this may be why Dave Thomas wrote: *J2EE desperately needs an industry standard Aspect Library to provide a simpler programming model [59, slide 13].*

8. RELATED WORK

Three closely related and complimentary research ideas are run-time weaving, load-time weaving and virtual machine support for aspect-oriented programming. We will discuss these ideas in detail below.

8.1 Run-time and Load-Time Weaving Approaches

There are several approaches for run-time weaving such as PROSE [47], Handi-Wrap [5], Eos [50], etc. A typical approach to runtime weaving is to attach hooks at all join points in the program at compile-time. The aspects can then use these hooks to attach and detach at run-time. An alternative approach is to attach hooks only at potentially interesting join points. In the former case, aspects can use all possible join points, excluding those that are created dynamically so the system will be more flexible. The disadvantage is the high overhead of unnecessary hooks. In the latter case, only those aspects that utilize existing hooks can be deployed at run-time, but the overhead will be minimal for a runtime approach.

Eos uses the second model, i.e. only instrument the join points that may potentially be needed. Handi-Wrap uses the first model, making all join points available through wrappers. PROSE indirectly uses the first model, exposing all join points through the debugger interface. PROSE allows aspects to be loaded dynamically without restarting the system. An additional advantage of indirectly exposing join points through debugger interface is that new join points (created by reflection) are registered automatically.As observed by Popovici et al. [47] and Ortin et al. [40], however, performance in both cases is a problem.

A load-time weaving approach delays weaving of crosscutting concerns until the class loader loads the class file and defines it to the virtual machine [31]. Load-time weaving approaches typically provide weaving information in the form of XML directives or annotations. The aspect weaver then revises the assemblies or classes according to weaving directives at load-time. Often a custom class loader is needed.

There are load-time weaving approaches for both Java and the .NET framework. For example, AspectJ [23] recently added load-time weaving support. Weave.NET [28] uses a similar approach for the .NET framework. The JMangler framework can also be used for load-time weaving [26]. It provides mechanisms to plug-in class-loaders into the JVM.

A benefit of the load- and run-time weaving approaches is that they delay weaving of aspect-oriented programs. It may be possible to improve incremental compilation using these approaches, although we do not currently have any evidence to confirm or to deny. A contribution of our approach might also be perceived as delaying weaving, however, we view the interface and corresponding contracts between the language designs and execution model designs as the main contribution of our work. The load-time weaving approaches do not provide these benefits.

8.2 Aspect Support in Virtual Machine

Steamloom [6] and *PROSE2* [46] both aim to achieve an aspect-aware Java Virtual Machine, to enhance the runtime performance of AOP. Steamloom extends the Jikes Research Virtual Machine (RVM), an open source Java virtual machine [1]. Traditional approaches for supporting dynamic crosscutting involve weaving aspects into the program at compilation. Steamloom moves weaving into the Virtual Machine (VM), which allows preserving the original structure of the code after compilation and shows performance improvements of 2.4 to 4 times when compared to AspectJ. It accomplishes this by modifying the Type Information Block to point methods to a stub that modifies the existing bytecode to weave in the advice. On the other hand, PROSE2 proposes an enhanced implementation for the original PROSE approach, by incorporating an execution monitor for joint points into the virtual machine. This execution monitor is then responsible for notifying the AOP engine which in turn executes the corresponding advices

Our approach and Steamloom are in some sense complimentary. Similar to Steamloom, our approach also advocates support for crosscutting in the execution models. Steamloom investigates techniques to improve the performance of these crosscutting mechanisms provided by the execution model, whereas, our approach focuses on separating the compiler implementations and execution model implementations by defining an interface between the two. Our focus is on providing the basic mechanisms at the interface that can be used as primitives by compiler implementations. Our approach thus potentially allows multiple language models to use the same VM and/or multiple VMs. Each of these VMs may have their own method of weaving. With respect to PROSE2, our approach does not require an extension to the virtual machine.

Steamloom and PROSE2, however, restrict the type hierarchy of aspects. An aspect must inherit from a special class. In languages like Java, this restriction burns the only available inheritance link. Our approach does not impose any restrictions on programming language constructs, leaving those design decisions to programming language designers and compiler implementers.

8.3 Other Related Work

Masuhara and Kiczales's work on modeling four aspect-oriented programming mechanism is related to the results that we demonstrated in Section 6. They model AspectJ, Adaptive Programming, HyperJ, and Open Classes. The main idea is to provide a semantics for these mechanisms by modeling the weaving process. Our work does not claim to provide any formal semantics for these aspect-oriented mechanisms; rather we show that a simple primitive can model these techniques. Our work can be a basis of developing a formal semantics for aspect-oriented languages. One could start by developing the precise semantics of the *Bind* invocation mechanism. The semantics of the *Bind* and the model of these aspect-oriented techniques in terms of bind can then be used to develop an operational semantics [45] for these mechanisms.

9. CONCLUSION

In this paper, we presented the design, implementation and evaluation of an aspect-oriented programming model that we call *Nu*. *Nu* provides *Bind* as an invocation mechanism for preserving design modularity in object code. The new invocation mechanism improves the conceptual integrity of the aspect-oriented programming models. Various advising and runtime properties of aspect-like constructs such as static, runtime, and instance-level advising can be implemented using *Bind* and simple patterns. We further showed that common advising structures in the HyperJ, Adaptive Programming, Composition Filters, AspectJ and Eos models can also be implemented in terms of this new invocation mechanism. This advance expands the useful program design space to include important styles: in particular, one involving the use of more then one language model to separate crosscutting concerns in a system.

We hypothesize, but haven't systematically investigated, that our approach promises to solve many other problems in AO approaches today such as compatibility with the existing tool chain, better run-time performance, cross AO-language compatibility, improved pointcut expressivity, efficient run-time weaving support, etc. The decoupling between language compilers and the virtual machine achieved by the interface provided by our invocation mechanism also has the potential to enable independent research in these areas. Simpler aspect language designs and compiler implementations might be realized without spending significant time on the optimization of the underlying AO execution models. Novel optimization mechanisms for the underlying execution models can be developed independent of the language design as long as it conforms to the interface.

[Notes to reviewers: The current version of Nu and the complete code for the examples presented in this paper are freely available from http://www.cs.iastate.edu/∼nu/]

10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers of the SPLAT 2006 workshop for their helpful comments. The author Hridesh Rajan would like to thank Gary T. Leavens, John Lefor and Eric Van Wyk for helpful discussions, Kevin J. Sullivan for comments on the early draft of the workshop version of this paper, and Gregor Kiczales for raising the virtual machine adoption issue.

11. REFERENCES

[1] The Jikes research virtual machine (RVM). http://jikesrvm.sourceforge.net/.

[2] The shared source common language infrastructure (SSCLI). http://research.microsoft.com/sscli/.

[3] M. Aksit, K. Wakita, J. Bosch, L. Bergmans, and A. Yonezawa. Abstracting Object Interactions Using Composition Filters. In R. Guerraoui, O. Nierstrasz, and M. Riveill, editors, *Proceedings of the ECOOP'93 Workshop on Object-Based Distributed Programming*, volume 791, pages 152–184. Springer-Verlag, 1994.

[4] R. Altman, A. Cyment, and N. Kicillof. On the need for setpoints. In *EIWAS 2005: European Interactive Workshop on Aspects in Software*, http://prog.vub.ac.be/events/eiwas2005/, 2005.

[5] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 86–95, New York, NY, USA, 2002. ACM Press.

[6] C. Bockisch, M. Haupt, M. Mezini, and K. Ostermann. Virtual machine support for dynamic join points. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 83–92, New York, NY, USA, 2004. ACM Press.

[7] J. Bonér. What are the key issues for commercial AOP use: how does AspectWerkz address them? In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 5–6, New York, NY, USA, 2004. ACM Press.

[8] G. Bracha and W. Cook. Mixin-based inheritance. In *OOPSLA/ECOOP '90: Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, pages 303–311, New York, NY, USA, 1990. ACM Press.

[9] F. P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison Wesley, Reading, Mass., second edition, 1995.

[10] A. Bryant and R. Feldt. AspectR - simple aspect-oriented programming in Ruby, Jan 2002.

[11] B. Burke. Aspect-oriented annotations. http://onjava.com article dated Aug 25, 2004.

[12] A. Campailla, S. Chaki, E. Clarke, S. Jha, and H. Veith. Efficient filtering in publish-subscribe systems using binary decision diagrams. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 443–452, Washington, DC, USA, 2001. IEEE Computer Society.

[13] A. Colyer and A. Clement. Large-scale aosd for middleware. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 56–65, New York, NY, USA, 2004. ACM Press.

[14] J. Dingel, D. Garlan, S. Jha, and D. Notkin. Reasoning about implicit invocation. *SIGSOFT Software Engineering Notes*, 23(6):209–21, Nov. 1998.

[15] ECMA. *Standard-334: C# Language Specification*, 2002.

[16] T. Elrad, R. E. Filman, and A. Bader. Aspect-oriented programming: Introduction. *Commun. ACM*, 44(10):29–32, 2001.

[17] Y. Endoh, H. Masuhara, and A. Yonezawa. Continuation join point. In C. Clifton, R. Lammel, and G. Leavens, editors, *In Foundations of Aspect-Oriented Languages workshop (FOAL 06), A workshop affiliated with AOSD 2006*, mar 2006.

[18] Eos web site. http://www.cs.iastate.edu/∼eos.

[19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[20] J. Gosling, B. Joy, and G. L. Steele. *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1996.

[21] J. Gough and D. Corney. Reading and writing PE-files with PERWAPI. Technical report, Queensland University of Technology, Brisbane, Australia, Sep 2005.

[22] R. Hirschfeld. Aspects - aspect-oriented programming with squeak. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 216–232, London, UK, 2003. Springer-Verlag.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *ECOOP 2001 — Object-Oriented Programming 15th European Conference*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer-Verlag, Budapest, Hungary, June 2001.

[24] G. Kiczales, J. Lamping, C. V. Lopes, C. Maeda, A. Mendhekar, and G. Murphy. Open implementation design guidelines. In *Proceedings of the 19th International Conference on Software Engineering*, pages 481–90, Boston, Massachusetts, 17–23 May 1997. IEEE.

[25] G. Kiczales and M. Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *ECOOP 2005 - Object-Oriented Programming, 19th European Conference, Glasgow, UK, July 25-29, 2005, Proceedings*, volume 3586 of *Lecture Notes in Computer Science*, pages 195–213. Springer, 2005.

[26] G. Kniesel, P. Costanza, and M. Austermann. Jmangler-a framework for load-time transformation of java class files. In *1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM 2001), 10 November 2001, Florence, Italy*, pages 100–110. IEEE Computer Society, 2001.

[27] S. Kojarski and D. H. Lorenz. Pluggable aop: designing aspect mechanisms for third-party composition. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 247–263, New York, NY, USA, 2005. ACM Press.

[28] D. Lafferty and V. Cahill. Language-independent aspect-oriented programming. In *OOPSLA '03: Proceedings of the 18th annual ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 1–12, New York,

NY, USA, 2003. ACM Press.

[29] J. Lamping. The role of base in aspect-oriented programming. In C. V. Lopes, A. Black, L. Kendall, and L. Bergmans, editors, *Int'l Workshop on Aspect-Oriented Programming (ECOOP 1999)*, June 1999.

[30] N. Lesiecki. Applying AspectJ to J2EE application development. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, New York, NY, USA, 2005. ACM Press.

[31] S. Liang and G. Bracha. Dynamic class loading in the java virtual machine. In *OOPSLA '98: Proceedings of the 13th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 36–44, New York, NY, USA, 1998. ACM Press.

[32] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-Oriented Programming with Adaptive Methods. *Commun. ACM*, 44(10):39–41, 2001.

[33] K. Lieberherr, C. Xiao, and I. Silva-Lepe. Propagation patterns: Graph-based specifications of cooperative behavior. Technical Report NU-CCS-91-14, Northeastern University, September 1991.

[34] K. J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Co., Boston, MA, USA, 1995.

[35] T. Lindholm and F. Yellin. Addison-Wesley, Reading, MA, USA, 1997.

[36] H. Masuhara and G. Kiczales. Modeling crosscutting in aspect-oriented mechanisms. In L. Cardelli, editor, *ECOOP 2003—Object-Oriented Programming, 17th European Conference*, volume 2743, pages 2–28, Berlin, July 2003.

[37] M. Mezini and K. Ostermann. Conquering aspects with caesar. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 90–99, New York, NY, USA, 2003. ACM Press.

[38] Microsoft Corporation. *Microsoft .NET*, 2001. URL: http://www.microsoft.com/net.

[39] D. Orleans and K. Lieberherr. Dj: Dynamic adaptive programming in java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns* , Kyoto, Japan, September 2001. Springer Verlag.

[40] F. Ortin and J. M. Cueva. Dynamic adaptation of application aspects. *Journal of Systems and Software*, 71(3):229–243, 2004.

[41] H. Ossher and P. Tarr. Multi-dimensional separation of concerns using hyperspaces. IBM Research Report 21452, IBM, Apr. 1999.

[42] J. Palsberg, B. Patt-Shamir, and K. Lieberherr. A new approach to compiling adaptive programs. In H. R. Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, 1996. Springer Verlag.

[43] J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[44] J. Pereira, F. Fabret, F. Llirbat, and D. Shasha. Efficient matching for web-based publish/subscribe systems. In *CooplS '02: Proceedings of the 7th International Conference on Cooperative Information Systems*, pages 162–173, London, UK, 2000. Springer-Verlag.

[45] G. D. Plotkin. A Structural Approach to Operational Semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

[46] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for java, 2003.

[47] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *AOSD '02: Proceedings of the 1st international conference on Aspect-oriented software development*, pages 141–147, New York, NY, USA, 2002. ACM Press.

[48] H. Rajan. *Unifying Aspect- and Object-Oriented Program Design*. PhD thesis, The University of Virginia, Charlottesville, Virginia, Aug. 2005.

[49] H. Rajan, R. Dyer, Y. Hanna, and H. Narayanappa. Preserving separation of concerns through compilation. In L. Bergmans, J. Brichau, and E. Ernst, editors, *In Software Engineering Properties of Languages and Aspect Technologies (SPLAT 06), A workshop affiliated with AOSD 2006*, mar 2006.

[50] H. Rajan and K. J. Sullivan. Eos: instance-level aspects for integrated system design. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, New York, NY, USA, 2003. ACM Press.

[51] H. Rajan and K. J. Sullivan. Need for instance level aspect language with rich pointcut language. In L. Bergmans, J. Brichau, P. Tarr, and E. Ernst, editors, *SPLAT: Software engineering Properties of Languages for Aspect Technologies*, mar 2003.

[52] H. Rajan and K. J. Sullivan. Classpects: unifying aspect- and object-oriented language design. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 59–68, New York, NY, USA, 2005. ACM Press.

[53] D. Sabbah. Aspects: from promise to reality. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 1–2, New York, NY, USA, 2004. ACM Press.

[54] K. Sakurai, H. Masuhara, N. Ubayashi, S. Matsuura, and S. Komiya. Association aspects. In *AOSD '04: Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 16–25, New York, NY, USA, 2004. ACM Press.

[55] O. Spinczyk, A. Gal, and W. Schroeder-Preikschat. AspectC++: an aspect-oriented extension to the c++ programming language. In *CRPITS '02: Proceedings of the Fortieth International Confernece on Tools Pacific*, pages 53–60, Darlinghurst, Australia, Australia, 2002. Australian Computer Society, Inc.

[56] K. J. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methodology*, 1(3):229–68, July 1992.

[57] P. Tarr and H. Ossher. Hyper/J user and installation manual. Technical report, IBM T. J. Watson Research Center, 2000.

[58] P. Tarr, H. L. Ossher, W. H. Harrison, and S. M. Sutton, Jr. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the 21st International Conference on Software Engineering*, May 1999.

[59] D. Thomas. Transitioning aosd from research park to main street, 2005. General Chair-Mira Mezini and Program Chair-Peri Tarr.