Project Summary
Aspect Interfaces

The popularity and adoption of Aspect-Oriented Software Development (AOSD) has naturally lead to an assessment of system modularity in the presence of aspects. The AOSD community has recently exposed issues with AOSD in four problem areas: ease of evolution, separate development, coupling and hiding. Before AOSD, crosscutting concerns were implicit in the code. AOSD has successfully made those concerns explicit in the code but it has left other issues, such as aspect-aspect and base-aspect dependencies, implicit. Our research proposes aspect interfaces (AIs) to make these issues explicit and to address the four problem areas mentioned above.

The broad definition of an interface is a convention used to allow communication between two software systems. AIs are conventions used to allow communication between aspects and base code or between aspects. We propose to include selector expressions in AIs. Selector expressions are a generalization of AspectJ pointcuts and Demeter strategies. They select nodes in an abstraction related to the base program, such as the execution tree of a program (AspectJ) or in objects (Demeter). More abstractly, selector languages are regular expressions over paths (including depth-first traversal paths) in graphs and their instances (obtained by unrolling the graphs).

Our approach to AIs is to express them using three components: base program abstractions (including abstractions of the data structures at run-time and abstractions of the program executions), selector expressions and predicates (over the base program abstractions and selector expressions). Abstractions of the base program include various graph structures such as class graphs, control flow graphs, abstract syntax trees and also both method signatures and event signatures.

AIs make it easier to extend an existing complex system with new behavior without having to know the details of the complex system. AIs are natural applications of the idea of information hiding to aspects. When writing an aspect, it is important that the aspect does not talk to too many classes in the base. And properly designed AIs protect the base from undesirable aspects.

To continue our work on AIs we will: (a) Develop a theory of AIs. (b) Study selector languages as suitable mechanisms to express AIs (both general purpose and concern-specific AIs) and their mappings. Study the theory of selector languages, both algorithmic lower and upper bounds for translation, decision and learning problems important to AIs. (c) Study techniques to build selector expressions interactively using learning algorithms. (d) Study the base evolution problem. As a base evolves we are concerned with its mapping to the AI and how this mapping changes. Which parts of the base evolution problem can be automated? (e) Integrate AIs with XAspects, a tool we have already built, to facilitate development of concern specific aspect languages.

Intellectual Merit

This research will address the fundamental problem for the growing AOSD community of how to best interface to aspects, both general purpose and concern-specific aspects. Our work will help the creation of AIs that can absorb changes. Our work produces concepts and tools for building concern-specific aspect languages and their corresponding AIs.

Broader Impact

Education is an important component of this proposal. We will develop a new undergraduate course on software development using AIs with tool support. In addition we will have continuing education workshops for professionals about design with AIs.

PROJECT DESCRIPTION

1 Project Background

Aspect-Oriented Software Development (AOSD), and Aspect-Oriented Programming (AOP), introduce the notion of an *aspect* as a mechanism that locally captures system concerns that would otherwise be distributed across the system's primary module decomposition (referred to as *base code*). Aspects define extra behavior called *advice* that is invoked at specific points during the program's execution called *join points*. The term *selector language* is used to refer to the sub-language used to define expressions that select join points.

The popularity and adoption of Aspect-Oriented Software Development and Aspect-Oriented Programming, has naturally lead to an assessment of aspect-oriented system modularity. The AOSD community [16, 1, 66, 70] has recently exposed issues with AOSD in four problem areas: ease of evolution, separate development, coupling and information hiding.

These issues are a manifestation of the non-modular characteristics of current AOP technologies. Aspects can bypass a module's interface introducing dependencies between the internal (to the module) data and the aspect. Aspects affect modules across the whole system defining communication channels between the aspect and the system's modules. Both the dependencies and communication channels introduced via aspects are explicit only from within the aspect's code; the remaining affected (by the aspect) modules are unaware of these dependencies.

Our research proposes aspect interfaces (AIs) to make these dependencies explicit and address the four problem areas mentioned above. We consider AIs for general purpose aspect languages, e.g., AspectJ, and concern-specific aspect languages, e.g., DemeterJ, COOL, RIDL. Our vision, as well as the vision of the original Northeastern PhD thesis work by Crista Lopes in this area [45], is that applications consist of multiple collaborating, concern-specific aspect languages. AIs will make the dependencies between aspects and components explicit allowing for separate development and their smooth symbiosis under evolution.

1.1 The Problem

In order to talk about modularity and how it can be observed we provide a definition of a modular implementation of a concern adopted from [16]:

- it is textually local,
- there is a well-defined interface that describes how it interacts with the rest of the system,
- the interface is an abstraction of the implementation, in that it is possible to make material changes to the implementation without violating the interface.
- an automatic mechanism enforces that every module satisfies its own interface and respects the interface of all other modules, and
- the module can be automatically composed – by a compiler, loader, linker etc. – via various configurations with other modules to produce a complete system.

We further take into account the five principles proposed by Meyer [49] that must be observed to ensure proper modularity.

Linguistic Modular Units Modules must correspond to syntactic units in the language used.

Few Interfaces Every module should communicate with as few others as possible.

Small Interfaces If any two modules communicate at all, they should exchange as little information as possible.

Explicit Interfaces Whenever modules *A* and *B* communicate, this must be obvious from the text of *A* or *B* or both.

Information Hiding All information about a module should be private to the module unless it is specifically declared public.

Current AOP technologies do not completely satisfy both the definition and/or the principles for modularity. In the case of a general purpose AOP language such as AspectJ, an aspect can be used in a way that it does not respect the interface of other modules. Aspects can expose to the rest of the program explicitly defined, private information of a module and the language's automatic mechanisms allow for this behavior. There is no hiding of information from the aspect by other modules. This forces aspect developers to understand large parts of the system in order to effectively add behavior with aspects. Finally, implementation alterations to what appears to be internal module information cannot be made by based on local reasoning. Instead further reasoning outside the module itself is needed to detect any applicable aspects and the effects the alterations may have on them and the system as a whole.

In the case of a concern specific aspect-oriented language, such as DemeterJ, there is an interface between the adaptive method and the static structure of the program (the program's *class graph*). However the interface is the whole class graph allowing for no information hiding and bringing about the same problems as in the case of general purpose aspect oriented languages.

There is no explicit agreement between aspects and the rest of the program modules on what is considered accessible by the aspect and what not. Currently everything is accessible to the aspects and nothing is hidden. The final decision of what is public and what is private for a given module after aspect composition is controlled by the aspect, which makes iterative program extensions difficult.

The asymmetric power over what is considered hidden and what not and the absence of an explicit interface that defines communication between aspects and program modules lowers aspect-oriented system modularity.

More concretely,

Ease of Evolution is decreased since aspects are written directly against the base program. Aspects will work for that specific base program but there is no explicit information kept about the abstract properties that the base program needs to fulfill.

Separate Development becomes more difficult to achieve since pointcuts inside aspects are written against the base program and use information (i.e. method and variable names, sequence

of method calls) specific to the base program's implementation. Concern-shy* selector languages do not completely resolve this problem (Section 1.3).

Coupling between aspects and (other) aspects, and, base program and aspects contribute negatively to both separate development and ease of evolution. Aspects attach extra behavior on any part of the base code, even on methods considered as internal to a module. This dependency is explicitly captured by the pointcut inside the aspect leaving the module itself unaware of this dependency. Structure-shyness provided by selector languages make dependency detection harder.

Information Hiding Program modules (other than aspects) have no guarantees that information defined by a module as private remains private. This limits information hiding and breaks existing module interfaces.

1.2 Interfaces

The broad definition of an interface is a convention used to allow communication between two software systems. Interfaces have been widely studied and used to practice information hiding. In [59], David Parnas says: "the purpose of hiding is to make inaccessible certain details that should not affect other parts of a system." The part that is made accessible is presented as an interface. Parnas' use of information hiding has been very influential and even today programming languages are designed to better support interface-oriented programming. In the proposed work we study aspect interfaces). AIs are different from API interfaces (signatures). An API interface is to functionality as an AI is to behavior. When an aspect satisfies an AI, the aspect is guaranteed to work when used with a base program and a mapping that connects the base program to the AI.

1.2.1 Information Hiding

Parnas' information hiding approach does not hide enough [31]. Public signatures made available through an interface should be used only sparingly. Sparing use of public signatures is dictated by the principle of low coupling. The Law of Demeter (LoD) [28], an example of a precise rule for coupling control is one approach on how to use public interfaces sparingly. An even better approach, based on traversal strategies, is to use some of the public signatures indirectly by specifying only a few of them explicitly and by using connectivity information to derive the rest. A recent tool, Prospector [48], uses this idea introduced back in 1992 [25]. This kind of structure-shyness† reduces the cost of API interface changes in languages like Java or C#.

AIs serve to protect an aspect from the details of a base program. The goal here is to make sparing use of base program information in the AI. In both cases, API interfaces and AIs, the issue of sparingly using information is important and indeed in both cases similar structure-shy mechanisms are used ([36] shows that, for example, the main selector languages of AspectJ and Demeter have equivalent expressiveness). Structure-shyness in this context means shy with respect to the structure of the base program.

*In general, a concern C is X-shy if the C implementation can adapt to small changes in X, or if the C implementation relies only on minimal information of X implementations. Domain-specific aspect languages allow problems to be decomposed to support concern-shy representations of concerns.

† We use the term structure-shy which in this context means shy with respect to the structure of the API interfaces.

Selector Language	Meta Graph	Instance Graph
AspectJ	call graph	tree of method calls
DemeterJ	class graph (is-a, has-a)	objects
Prospector	class graph (is-a, 0-argument methods)	path of method calls

Table 1: Selector languages with their meta graphs and their instances.

Aspect-oriented programming without AIs has several drawbacks:

1. accidental complexity (and thus cost) in the expression of aspect code (since it must be written against arbitrarily complex base code).
2. significantly increased cost of change due to typically extensive dependences of aspect code on implementation details of the base code. Alterations made to the base code render aspects in-applicable or cause erroneous behavior without any compile time or runtime errors.
3. loss of concurrency in development due to the dependencies between aspects and the base on any of the base programs properties (private or public).

AIs play an important role in aspect-oriented programming but they have not been widely researched.

1.3 Selector languages

Selector languages allow for the definition of expressions that select nodes and edges in graphs that are abstractions related to a program. Selector languages are used in several tools, for example, in tools for Aspect-Oriented Programming (AOP) (AspectJ [15], Caesar [50]), in tools for Adaptive Programming (AP) (JAsCo[68], DJ [54]) and in tools for API programming (Persephone [3], Prospector [48]). Different communities give different names to selector languages: crosscut languages or pointcut languages in AOP , traversal specifications or traversal strategies in AP, traversal specifications or queries in API-Programming.

A selector language operates in the space of graphs, called meta graphs, and instances of those graphs, called instance graphs (Table 1). A selector language expression has a source node and selects a set of nodes or edges for a given instance graph that conforms to some meta graph.

Selector languages, although overall beneficial, give rise to some complications when the meta graph is altered. The meta graphs will evolve and this requires a careful testing of the selector expressions to make sure they still select the right paths. For example, in an AspectJ program, the renaming of a few private method names in the base program may lead to a broken aspect, e.g., the aspect may never select a join point. Currently, AspectJ does not check for the unsatisfiability of pointcuts, except for trivial cases. When using selector languages in practical tools, several algorithmic problems need to be solved:

Translation Problems We need to implement selector languages and this means translating them to lower level languages or interpreting them. Given a meta graph and a selector expression, we need to produce an efficient function that takes an instance graph as input and produces as output a marked instance graph with the selected nodes marked. Given a meta graph and

a selector expression, we need to compile them into an efficient structure that allows us to explore an instance tree at run-time efficiently.

Decision Problems Our first attempt in formally defining some important decision problems has led to interesting results [36]. The selector satisfiability problem is important for checking whether an AI holds for a base program. Useful decision problems include: Satisfiability, Implication, Always, Never, Empty-Intersection etc. [36].

Learning Given a meta graph and examples of marked meta graphs that show the effect of the sought after selector expression, construct a minimal selector expression that produces the given marked meta graphs.

1.4 PIs' Prior Accomplishments on Aspect Interfaces

The term *aspect interface* has been introduced in the first PhD thesis dedicated to AOSD issues by Crista Lopes [45] (advised by the PI and co-advised by Gregor Kiczales then at PARC). The thesis introduces the concern-specific aspect languages COOL and RIDL and discusses their AIs.

The PI has worked on AIs since the mid 90's as part of his work on Adaptive Programming [25]. The initial focus was on adaptive components whose interfaces put constraints on the class graphs through path constraints and cardinality constraints [27] (section 8.5). This work was followed by an OOPSLA 1998 paper with Mira Mezini on adaptive plug-and-play components [51] that discussed the concept of interfaces for adaptive components. With David Lorenz and Mira Mezini, we generalized the interfaces to aspects [32]. This work has influenced several aspect systems, such as Caesar and ObjectTeams. Johan Ovlinger's PhD thesis in the PI's group [56] and a related journal paper with David Lorenz [33] have proposed an approach to simple AIs.

The co-PI joined the investigation of selector languages in 2003 and the collaboration has resulted in one FOAL workshop publication [36] and one journal submission.

PI Lieberherr is a leader in the AOSD field. He is on the steering committee of AOSA (Aspect-Oriented Software Association) and was Program Committee Chair of the Aspect-Oriented Software Development Conference (AOSD 2004). He was a keynote speaker at ICSE 2004.

1.5 Related Work

1.5.1 Interfaces for aspects

Recent attempts to resolve modularity issues in AOSD evolve around the notion of an interface [16, 1, 66] between aspects and base code components.

Kiczales and Mezini [16] advocate that in the presence of aspects, a module's interface has to further include pointcuts from aspects that apply to the module in question. These augmented interface definitions, named *aspect-aware interfaces*, can only be determined after the complete configuration of the system's components is known. Aspect-aware interfaces do not provide any extra information hiding capabilities to the base program's modules.

Open Modules [1], extend the traditional notion of a software module to include in its interface pointcut specifications. In this way a module can export, and as such make publicly available, pointcuts within its implementation. This approach gives a balanced control between module and aspect developers in terms of information hiding thus allowing for separate (parallel) evolution of

aspects and modules on the agreed upon interface. The interface of a crosscutting concern can affect multiple modules at different join points on each one. Thus an aspect's interface is sprinkled along module interfaces and not localized making it harder (if not impossible at times) for aspect developers to develop their aspects.

Kevin Sullivan and Bill Griswold and their students [66] advocate an XPI (crosscutting program interface) as a means to achieve separate development and explicit dependencies between implementations of crosscutting concerns and base code. XPIs at present are a design artifact where the agreed upon interface between aspects and base code is explicitly stated using English. Although XPIs assist both during design and development, there is no mechanical checking available to verify the implementation against the agreed upon interface.

Kiczales and Mezini in [17] discuss the benefits of using different programming language mechanisms (procedures, annotations, advice and pointcuts) used to provide separation of concerns at the code level. The resulting guidelines from their analysis sketch the situations where each mechanism will be most effective. The inherent modularity issues associated with each technology are not addressed.

1.5.2 Selector languages

The pioneering work in selector languages are query languages for object-oriented databases. In databases, the meta graph is an object schema, the instances are objects and the selector language is a query language. Using selector languages for implementing "Have X, Want Y" concerns was pioneered by the Demeter work [25] for object navigation and by the Persephone work (Allemang et al. [3]) for API navigation.

In [48] David Mandelin with coauthors at Berkeley and IBM have independently rediscovered the value of "Have X, Want Y" concerns. They developed a Persephone-like tool, called Prospector, using a shortest path semantics. While in Persephone the selector language is expressive enough to choose the desired path, Prospector let's the programmer choose from a set of suggested alternatives.

AspectJ [15, 67] uses a selector language for choosing points in the execution of a Java program. AspectJ works with paths from the root of the execution tree. Robert Walker[69] has extended the language to reason about the history of the entire computation. This corresponds to reasoning about depth-first traversal paths and not only simple paths from the root.

The interest in selector languages is growing. At OOPSLA 2005 there will be one session on selector languages: two papers from Stanford and one from the AspectBench Compiler for AspectJ(abc) team at Oxford, McGill and BRICS in Denmark [2]. The abc team paper builds on Robert Walker's work by working with histories. The selector expression language has two layers: In the first layer they define interesting symbols using AspectJ's pointcut language and in the second layer they use regular expressions that picks points in the execution tree.

The *Shadow Programming* technique [71], particularly, the *Pointcut Evaluator* facility, makes aspects less coupled to syntactic properties of base programs, but more to their semantic properties. This join point selection mechanism will more likely deliver reusable and evolveable aspects, while traditional syntax based join point selection mechanisms have been criticized to be fragile due to the syntax's volatility.

Researchers have explored the usage of Prolog as a selector language, taking advantage of the language's unification features to allow for more robust selector expressions.

Ostermann *et al.* [55] propose the ALPHA language, allowing the user to write Prolog queries over four models: the abstract syntax tree, object store (heap), static types of all expressions, and trace of the program. Queries in this form can bind variables and refer to these; hence, with the ability to query the program trace, a pointcut can build a history context. The authors argue that this type of pointcut decreases complexity while increasing expressiveness

Klose and Ostermann [18] propose a Prolog-based selector language over a program's **complete** execution trace. A predicate can refer to events in the computation's steps that are to follow. Each program event holds a unique timestamp allowing for temporal relations between events to be captured in selector expressions. Advice is iteratively applied until a fix-point is reached (if it exists, else the computation diverges). A fix-point for an advice is reached when the set of events that trigger the advice remains the same between two consecutive applications of the advice's code. The current implementations lack in static analysis tools that can assist in the evaluation of selector expressions and can improve efficiency.

In [6], Eichberg, Mezini and Ostermann use the XQuery language as a selector language while the underlying model is an XML representation of program information (translated from class files using a special tool). Being a Turing-complete language, XQuery enables more expressive selections of join points, which are not supported by AspectJ. However, using XQuery complicates static analysis tasks due to its undecidable nature. Meanwhile, not all programmers are comfortable with an XML representation of program information.

LogicAJ [60, 19] by Günter Kniesel and Tobias Rho is a uniformly generic aspect language making use of logic meta-variables and the concept of unification to make aspects less coupled to syntactic properties of base programs, and thus to promote aspect reusability and evolution. It can simulate part of the functionalities provided by statically executable advice [35] and pointcut evaluator facilities [71].

2 Proposed Research

To continue our work on aspect interfaces we will

1. Incorporate aspect interfaces into existing tools for both generic aspect-oriented languages as well as concern-specific aspect languages and study their effects to system modularity.

2. Study selector languages as suitable mechanisms to express aspect interfaces (both general purpose and concern-specific aspect interfaces) and their mappings. Study the theory of selector languages, both algorithmic lower and upper bounds for translation, decision and learning problems important to AIs. We will develop the Selector Library to facilitate the implementation of selector languages.

3. Study techniques to build selector expressions interactively using learning algorithms that can assist developers in defining robust selector expressions that capture the developer's intentions.

4. Study the base evolution problem. As a base evolves we are concerned with its mapping to the aspect interface and how this mapping changes, explore the possibility to automate such changes.

5. Integrate aspect interfaces with XAspects [61, 62], a tool we have already built, to facilitate development of concern specific aspect languages. We will develop an informal method to translate requirements into concern-specific aspect languages and corresponding selector languages. We will develop style rules for concern-specific aspect languages, continuing our work that started with the Law of Demeter almost 20 years ago. Recent results [53] have provided a generalization of the Law of Demeter to aspects.

2.1 Aspect Interfaces

We demonstrate how aspect interfaces address the three issues: ease of evolution, parallel development and coupling through two examples, for a concern-specific aspect language (DemeterJ) and for a generalized aspect language (AspectJ).

2.1.1 Traversal Aspect Example

Our example is about systems of equations in which we want to check that all used variables are defined (we call this a *semantic checker*). A simple equation system could be $x = 5; y = 9; z = x + y;$. The implementation of the semantic checker should be unaffected by any changes made to the equation system that do not alter the definition of used and defined variables, i.e, change to infix notation, adding new operators etc.

Therefore, we introduce a concern-specific aspect interface against which we can program the semantic checker in such a way that it is shielded from details such as infix versus prefix operators. The AI consists of a class graph, called an interface class graph (**icg**), traversal strategies and constraints imposed on these traversal strategies. Figure 1 shows the complete aspect interface along with a diagrammatical representation (in UML) of the AI's interface class graph.

The **icg** serves as a constraint on any concrete implementation of the `ExpressionICG` AI in order for the strategies defined in `ExpressionICG` to be applicable. Furthermore, the constraints defined in the aspect interface impose further requirements that need to be met by any concrete implementation of this aspect interface. The first constraint expresses that each definition introduces exactly one name. Without the nonempty constraints, the adaptive program is meaningless. The non-empty constraints state that there must be at least one path in the interface class graph that satisfies the strategy. The predicates `unique` and `nonempty` are provided by the aspect language, i.e. DemeterJ.

Figure 2 gives an example implementation of the `ExpressionICG` aspect interface (on the right) along with a visitor implementation and a driver class (on the left). `InfixEQSystem` defines the structural relations between concrete classes. In the cases where a class graph implements an aspect interface, mappings between the concrete class graph classes to the aspect interface classes are also provided. The concrete class graph (`InfixEQSystem`) provides a definition of its equation system and a mapping M between the entities in its class graph and the entities in the interface class graph. The mapping allows for classes to be mapped to classes but also strategies to be mapped to strategies. The special method **traverse** on the `InfixEQSystem` class is provided by DemeterJ and can be used to invoke adaptive methods. The `traverse` method takes as arguments one of the strategies in `InfixEQSystem` and a visitor that will invoke its methods upon reaching an instance that has the same type as the visitor method's argument. The name of the visitor method indicates as to whether this happens before traversing the object or after.

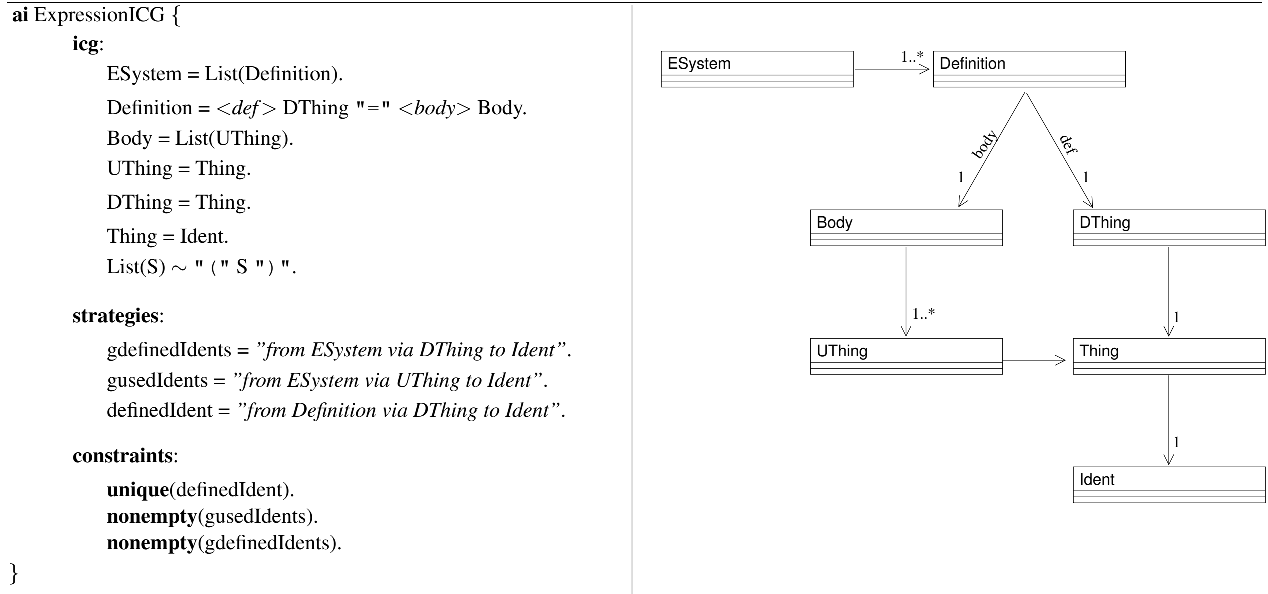

Figure 1: The full definition of the aspect interface includes an interface class graph, the strategies on the interface class graph and constraints on the strategies. The UML diagram is equivalent to the interface class graph defined in `ExpressionICG`.

At compile-time, DemeterJ will do the following:

1. use this mapping M to ensure that the class graph in `InfixEQSystem` satisfies the interface class graph from `ExpressionICG`, i.e., for each edge e connecting two nodes n_1 and n_2 in the interface class graph the mapped edge $M(e)$ connects the mapped nodes $M(n_1)$ and $M(n_2)$ in the class graph.

2. the strategies defined in `ExpressionICG` will be expanded according to the mapping and made available through `InfixEQSystem` class.

3. each of the constraints will be validated on the expanded strategies for `InfixEQSystem`.

If any of the above steps fails then an error is issued stating that `InfixEQSystem` does not implement `ExpressionICG`.

As a first evolution step we want to change from infix notation to prefix notation. Moving to a prefix notation requires to change the definition of `Compound` in `InfixEQSystem` to

```
Compound = <op> Op <lrand>List(Expression) <rrand>List(Expression)
```

This change does not affect the aspect interface at all. We update the equation system class graph and keep the old mapping. All constraints of the aspect interface are still satisfied after they are mapped into the actual interface class graph and the adaptive method will function correctly.

It is important to note that during this evolution step, only the aspect interface and the concrete implementation of the interface class graph was needed. Furthermore, the static assurances provided by the tool because of the aspect interface did not require re-testing of the driver code to ensure that the strategies pick the correct traversals and that the semantic checker still operates as

```
class Main {
  public static void main(String[] args){
    Visitor v = new IDPrintVisitor();
    System.out.println("IDs in def:");
    InfixEQSystem.traverse(gdefinedIdents,v);
    System.out.println("IDs in use:");
    InfixEQSystem.traverse(gusedIdents,v);
  }
}

class IDPrintVisitor extends Visitor {
    public void before(Ident id) {
        System.out.println(id);
    }
}
```

```
cd InfixEQSystem implements ExpressionICG {
  EquationSystem = <equations> List(Equation).
  List(S) ~ "(" {S} ")".
  Equation = <lhs> Variable "=" <rhs> Expression.
  Expression : Simple | Compound.
  Simple : Variable | Numerical.
  Variable = Ident.
  Numerical = <v> int.
  Compound =<lrand>List(Expression) <op> Op <
      rrand>List(Expression).
  Op : Add.
  Add = "+".

  use EquationSystem as ESystem,
      Equation as Definition,
      (−>∗,lhs,∗ to Variable) as DThing,
      (−>∗,rhs,∗ to Variable) as UThing,
      Variable as Thing.
}
```

Figure 2: `InfixEQSystem` defines a class graph and a mapping of the entities in the class graph to the interface class graph of `ExpressionICG`. The driver class `Main` uses the strategies defined in `ExpressionICG` and the `IDPrintVisitor`.

expected. The aspect interface allows in this case for separate development and ease of evolution. Hiding irrelevant information through the aspect interface provides for higher system modularity.

In the next evolution step we want to add the capability to define functions with one argument. A simple equation system with function definition and use can be $f(x) = x + 89; z = f(5)$; This evolution step affects information that is relevant to the semantic checker. The semantic checker has to also deal with parameter names on each function definition but also usages of function definitions that may appear on the right-hand side of equations. A naive approach would be to alter the class graph definition to accommodate for function definitions to appear on the right hand side of equations, i.e.,

```
Equation = <lhs> DExpression ``='' <rhs> Expression.
DExpression : Variable | ParametricVariable.
ParametricVariable = Variable ``('' Variable ``)''.
```

Without altering the mapping and leaving the aspect interface intact this approach will result in a compile time error. The predicate `unique(definedIdent)` can no longer be satisfied. For this evolution step the interface has to be changed (Figure 3). With a new interface class graph `ParamExprICG` we can abstractly reason about semantically checking systems with simple functions. To make the constraint hold again, we strengthen the strategy by adding an extra directive "`via Thing`" to `definedIdent`. Because `definedIdent` is a sub-strategy of `gdefinedIdents`, we need to update `gdefinedIdents` in the same way. As a final step we update the adaptive methods as shown in Figure 4.

In this evolution step, the aspect interface helped by disallowing a naive extension that would violate the intended behavior of the original aspect interface. The nature of the evolution required

```
ai ParamExprICG {
  icg:
      ESystem = List(Definition).
      Definition = <def> DThing "=" <body> Body.
      Body = List(UThing).
      UThing = Thing ["(" <aparam> UParamName ")"].
      DThing = Thing["(" <fparam> DParamName ")"].
      Thing = Ident.
      UParamName = Ident.
      DParamName = Ident.
      List(S) ~ "(" S ")".
  strategies:
      gdefinedIdents = "from ESystem via DThing via Thing to Ident".
      gusedIdents = "from ESystem via UThing to Ident".
      defParam = "from Definition via DParamName to Ident".
      definedIdent = "from Definition via DThing via Thing to Ident".
  constraints:
      unique(definedIdent).
      nonempty(gusedIdents).
      nonempty(gdefinedIdents).
      unique(defParam).
}
```

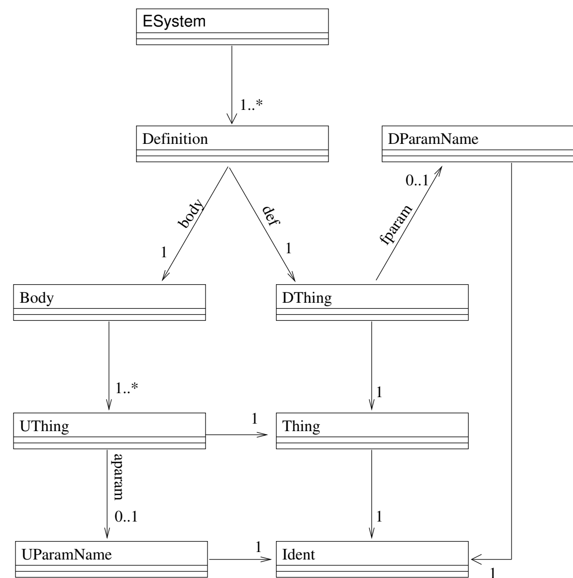

Figure 3: The evolved aspect interface (left) and the UML representation of the extended aspect interface class graph (right).

an extension of the interface and that resulted to changes in the driver class and a new class graph‡. It is important to note how the aspect interface exposed the erroneous usage of `ExpressionICG` interface for this evolution step and assisted in updating all the dependent components due to the definition of `ParamExprICG`.

The study of aspect interfaces started 10 years ago in the context of Demeter [27] (section 8.5), [52]. Even though DemeterJ, and Adaptive Programming in general, is a concern-specific aspect oriented system, there is a correspondence to general purpose aspect oriented systems. Using the DemeterJ terminology, traversal strategies and visitor method signatures correspond to AspectJ pointcuts and the visitor method bodies correspond to AspectJ advice. Although this is a special case of AOP where the join points only involve join points of a traversal (and not a general Java program as in AspectJ), key problems of aspect interfaces are already present. In fact recent work [55] uses ideas from both DemeterJ and AspectJ as mechanisms for providing a more expressive selector language in a general purpose aspect oriented programing language.

An important ingredient of aspect interfaces that we propose to research is that they use the proper abstractions to filter out the unimportant details in the base program and that they express predicates important to the proper functioning of the aspects. The predicates must be checkable ideally at compile-time but some may require run-time checking.

‡Not shown here due to space limitations

```
class Main {
  public static void main(String[] args){
    Visitor v = new IDPrintVisitor();
    GVisitor v1 = new GVisitor();
    System.out.println("defined IDs");
    ParamEQSystem.traverse(gdefinedIdents,v);
    System.out.println("undefined IDs with
        environment extension");
    ParamEQSystem.traverse(gusedIdents, v1);
  }
}
```

```
class IDPrintVisitor extends Visitor {
    public void before(Ident id) {
        System.out.println(id);
    }
}
class GVisitor extends Visitor {
    public void before(Definition d) {
      Ident dparam = (Ident) ParamEQSystem.fetch(d,
          defParam);
        System.out.println(" defined parameter: " +
            dparam);
    }
    public void before(Ident id) {
        System.out.println(id);
    }
}
```

Figure 4: Changes to the interface affect `Main`. The definition of `GVisitor` is used to check for the local parameter names in parametric equations.

2.1.2 Message Interface Example

We believe that capturing interesting events with AspectJ-style pointcuts is impractical for very large programs, and this is due to the tight coupling between the aspect and base program. For a motivating example, consider an email program and using aspect interfaces to support plugins. A plugin is simply an extension of an existing program that adds additional functionality, so it is natural to think of implementing one using AOP.

For instance, consider writing a plugin to check that messages containing the word *attach* actually have an attachment. The functional interface would have methods such as:

```
String getText(Message m)
String getRecipient(Message m)
File[] getAttachments(Message m)
...
```

and the event interface would export events such as:

```
sending(Message m)
receiving(Message m)
...
```

Both the functional and aspects interface are implemented by the base program. The former by writing methods, and the latter in a number of ways. So, to write the plugin we could write advice in AspectJ-style notation as:

```
before sending(Message m) {
  if (getText(m).contains("attach") &&
      getAttachments(m).length == 0) {
    // show an error . . .
  }
}
```

This advice looks almost exactly like AspectJ, but the key point is that the pointcut `sending` is expressed over the aspect interface and not the actual program. Some of the benefits of this approach are:

- Ease of evolution: A pointcut over an abstraction of the program – i.e. the aspect interface – is more resilient to change. Thus as the base program evolves much less effort will be spent maintaining the aspect code. In this example, even if the base program doesn't cleanly implement the *sending* event at a single point and changes this implementation constantly, aspects will not be affected because they are only attached to the `sending` pointcut in the aspect interface.

- Coupling: The aspect interfaces provide a clean abstraction that will ultimately help decouple aspects from base program. This abstraction provides better modularity and ease in the understanding and extending of complex programs, so that aspects can finally be realized on large-scale systems. If an aspect's pointcut were written over the functional API of this base program it would require knowing all the locations in the code where a message is sent, as well as how to construct the message context. Having the base program writer implement and export this pointcut decouples this program from the aspect program dramatically.

- Separate development: Since the aspect interface helps decouple aspects and the base program, multiple aspects can be implemented as the base program is developed; provided the base program ultimately conforms to this interface. For example, multiple plugins could be written using aspect interfaces that do very different things while the base program is being developed. This is very relevant in software development today. Many mainstream applications are being developed with a core base that most of the functionality provided by plugins. Enabling plugins to be developed independently of the base program via aspects would greatly help the efficiency and quality of these applications.

2.2 Selector Library and XAspects

In this part of our work we will automate the implementation of selector languages. In the third year of the grant, we hope to be able to generalize our experiences into a generic tool (Selector Library) that takes as input a suitably constrained syntactic and semantic definition of a selector language and that automatically or semi-automatically generates efficient algorithms for translation, decision problems and learning.

While general purpose aspect languages are useful, certain crosscutting concerns are best handled when using concern-specific aspect languages [45, 70]. A *concern-specific aspect language* is a custom language that allows special forms of crosscutting concerns to be decomposed into modularized constructs. Examples of concern-specific aspect languages include tools for dealing with coordination concerns, object marshaling, and class graph traversal.

XAspects presents a system that integrates concern-specific aspect languages with AspectJ using a technique that allows concern-specific language implementations to leverage a powerful subset of AspectJ. The concern-specific languages are integrated by using a plug-in architecture that allows the constructs created by the languages to cooperate with each other while not being aware that the other plug-ins exist.

We plan to integrate XAspects with the proposed Selector Library. The ability to define multiple different concern-specific languages within XAspects provides a suitable broad test bed for the development and evaluation of the Selector Library.

3 Time Line and Management Plan

In year one we start with a study of aspect interfaces by analyzing existing aspect-oriented programs. We will analyze the concern structure behind those programs and how interfaces for general purpose aspects as well as concern-specific aspects improve those programs. This will give us material for how to move from concern descriptions to concern-specific aspect languages and their selector languages and how to address the interactions between those aspect languages. We will develop style rules for concern-specific aspect languages. The existing XAspects will be a test bed to implement the concern-specific aspect languages. We will make progress on the base evolution problem and start the design of an improved XAspects with AIs.

In year two we will implement XAspects with AIs. In parallel we will develop a theory of AIs. The learning problem for selector expressions will be studied algorithmically and a tool for interactively learning selector expressions will be designed.

In year three we will design and implement the Selector Library. We will study how the Selector Library influences the process of developing concern-specific aspect languages. We will develop a method for analyzing requirements and translating concerns into concern-specific aspect languages.

4 Significant Impact

4.1 Social

AOSD is becoming quite popular thanks to the high quality of the AspectJ implementations. Now the community is ready for a study of aspect interfaces after aspects have become known and have been proven to be useful. Without a carefully designed aspect interface technology as proposed by the PIs there is a danger that too many aspect-oriented programs become hard to maintain and they might malfunction unexpectedly with high cost to software companies and software users. After having nurtured the AOSD field for 17 years, the PI Lieberherr feels the urge to work on this hurdle to AOSD.

4.2 Education

We will incorporate aspect interfaces in an undergraduate course on software development. This is in line with teaching interface-centric programming techniques. This grant will train 3 PhD students to push the state-of-the-art in software development. The PI Lieberherr has a good track record of successful PhDs (currently 12), e.g., Crista Lopes as the mother of AOSD and Ian Holland as the father of the Law of Demeter.

4.3 Results from Prior NSF Support for Karl Lieberherr

We briefly summarize the results from 4 previous NSF grants. Our work was influential in shaping the area of Aspect-Oriented Software Development (separation of class structure concern, navigation concern and Adaptive Visitor concern inspired COOL and RIDL [45] which in turn influenced AspectJ).

- NSF-Grant CCR-0098643, with David Lorenz (2001-2002).
  Title: Collaboration-oriented Aspects

  We improve previously exponential algorithms for compiling aspect-oriented programs to become polynomial time [20]. [34] shows a good way to combine aspects and modules. In [35] we show how to use AspectJ to check the Law of Demeter and outline how to improve AspectJ's static checking capabilities. In [62] we continue our earlier work on domain-specific aspect languages and provide a tool that we want to develop further in this proposal. [47] summarizes our ideas for better referential mechanisms in programming languages.

- NSF-Grant CCR-9402486 (Software Engineering) (1994-97). Cosponsored by DARPA.
  Title: Engineering Adaptive Software

  The key results of this grant are a formalization of the important concepts of adaptive programming [58], fast compilation algorithms [58, 57], an evolution framework for adaptive programming [11, 13], an application of adaptiveness to parameter passing in distributed systems [46], and the distribution of the Adaptive Programming tools through a undergraduate/graduate level textbook [27].

- NSF-Grant CCR-9102578 (1991-93).
  Title: Abstractions for Organizing Classes

  The key result of this grant is a new method to develop software based on separating the concerns: structure, traversal and traversal-based collaboration. Publications: [41, 43, 9, 29, 25, 37, 42, 38, 65, 27, 14, 64, 63, 4, 12, 30, 44].

- NSF-Grant MCS80-04490 (1980-1982), NSF-Grant MCS82-01878 (1982-1983).
  Title: Combinatorial Optimization and Search Problems

  The key contributions of this work are: 1. A seminal contribution to the state-of-the-art in hardware description languages which resulted in the invited paper [26]. 2. A theory of P-optimal approximation algorithms for combinatorial optimization problems. Publications: [24, 39, 40, 21, 22, 23, 7, 10, 8]

Acknowledgements: We would like to thank Therapon Skotiniotis for his contributions to the proposal. Thanks to Jeffrey Palm for his message interface section.

References cited

[1] Jonathan Aldrich. Open Modules:modular reasoning about advice. In *European Conference on Object-Oriented Programming*, 2005.

[2] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julie Tibble. Adding trace matching in AspectJ. In *Object Oriented Programming, Systems, Languages and Applications*, 2005.

[3] Dean Allemang and Karl J. Lieberherr. Softening Dependencies between Interfaces. Technical Report NU-CCS-98-07, College of Computer Science, Northeastern University, Boston, MA, August 1998.

[4] Paul L. Bergstein and Walter L. Hürsch. Maintaining behavioral consistency during schema evolution. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 176–193, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science.

[5] Ron Crocker and Guy L. Steele Jr., editors. *Companion of the 18thAnnual Conference on Object-Oriented Programming Systems, Languages, and Applications*, Anaheim, California, 2003. ACM Press.

[6] Michael Eichberg, Mira Mezini, and Klaus Ostermann. Pointcuts as functional queries. In Wei-Ngan Chin, editor, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004*, Lecture Notes in Computer Science, pages 366–382, Taipei, Taiwan, November 2004. Springer-Verlag Heidelberg.

[7] Jim Finn and Karl J. Lieberherr. Primality testing and factoring. *Theoretical Computer Science*, 23:211–215, 1983.

[8] Steven M. German and Karl J. Lieberherr. Zeus: A language for expressing algorithms in hardware. *IEEE Computer Magazine*, pages 55–65, February 1985.

[9] Ian M. Holland. Specifying reusable components using contracts. In *European Conference on Object-Oriented Programming*, pages 287–308, Utrecht, Netherlands, 1992. Springer Verlag Lecture Notes 615.

[10] M.A. Huang and Karl J. Lieberherr. Implications of forbidden structures for extremal algorithmic problems. *Theoretical Computer Science*, 40:195–210, 1985.

[11] Walter Hürsch. *Maintaining Consistency and Behavior of Object-Oriented Systems during Evolution*. PhD thesis, Northeastern University, 1995. 331 pages.

[12] Walter L. Hürsch, Karl J. Lieberherr, and Sougata Mukherjea. Object-oriented schema extension and abstraction. In *ACM Computer Science Conference, Symposium on Applied Computing*, pages 54–62, Indianapolis, Indiana, February 1993. ACM Press.

[13] Walter L. Hürsch and Linda M. Seiter. Automating the Evolution of Object-Oriented Systems. In *International Symposium on Object Technologies for Advanced Software*, pages 2–21, Kanazawa, Japan, March 1996. Springer Verlag, Lecture Notes in Computer Science.

[14] Linda Keszenheimer. Specifying and adapting object behavior during system evolution. In *Conference on Software Maintenance*, pages 254–261, Montreal, Canada, 1993. IEEE Press.

[15] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William Griswold. An Overview of AspectJ. In Jorgen Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.

[16] Gregor Kiczales and Mira Mezini. Aspect-oriented programming and modular reasoning. In *ICSE '05: Proceedings of the 27th International Conference on Software Engineering*, pages 49–58, New York, NY, USA, 2005. ACM Press.

[17] Gregor Kiczales and Mira Mezini. Separation of concerns with procedures, annotations, advice and pointcuts. In *European Conference on Object-Oriented Programming*, 2005.

[18] Karl Klose and Klaus Ostermann. Back to the future: Pointcuts as predicates over traces. In *Workshop on Foundations of Aspect-Oriented Languages (FOAL), AOSD*, 2005.

[19] Günter Kniesel, Tobias Rho, and Stefan Hanenberg. Evolvable Pattern Implementations Need Generic Aspects. In *Proceedings of Workshop on Reflection, AOP and Meta-Data for Software Evolution*, June 2004.

[20] Karl Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[21] Karl J. Lieberherr. Probabilistic combinatorial optimization. In J. Gruska and M. Chytil, editors, *Mathematical Foundations of Computer Science, Strbske Pleso, Czechoslovakia*, volume 118, pages 423–432. Springer Verlag, Lecture Notes in Computer Science, 1981.

[22] Karl J. Lieberherr. Uniform complexity and digital signatures. *Theoretical Computer Science*, 16:99–110, 1981.

[23] Karl J. Lieberherr. Uniform complexity and digital signatures. In *International Colloquium on Automata, Languages and Programming*, pages 530–543, Haifa, Israel, 1981. Springer Verlag, Lecture Notes in Computer Science.

[24] Karl J. Lieberherr. Algorithmic extremal problems in combinatorial optimization. *Journal of Algorithms*, 3:225–244, 1982.

[25] Karl J. Lieberherr. Component enhancement: An adaptive reusability mechanism for groups of collaborating classes. In J. van Leeuwen, editor, *Information Processing '92, 12th World Computer Congress*, pages 179–185, Madrid, Spain, 1992. Elsevier.

[26] Karl J. Lieberherr. The Hardware Description Language Zeus. *IEEE Design and Test of Computers*, pages 60–62, September 1992.

[27] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, web book at www.ccs.neu.edu/research/demeter.

[28] Karl J. Lieberherr and Ian Holland. Formulations and Benefits of the Law of Demeter. *SIGPLAN Notices*, 24(3):67–78, March 1989.

[29] Karl J. Lieberherr, Walter Hürsch, Ignacio Silva-Lepe, and Cun Xiao. Experience with a graph-based propagation pattern programming tool. In Gene Forte et al., editor, *International Workshop on CASE*, pages 114–119, Montréal, Canada, 1992. IEEE Computer Society.

[30] Karl J. Lieberherr, Walter L. Hürsch, and Cun Xiao. Object-extending class transformations. *Formal Aspects of Computing, the International Journal of Formal Methods*, (6):391–416, 1994. Also available as Technical Report NU-CCS-91-8, Northeastern University.

[31] Karl J. Lieberherr and David Lorenz. Coupling Aspect-Oriented and Adaptive Programming. In Robert Filman, Tzilla Elrad, Siobhán Clarke, and Mehmet Aksit, editors, *Aspect-Oriented Software Development*, pages 145–164. Addison-Wesley, 2004.

[32] Karl J. Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[33] Karl J. Lieberherr, David Lorenz, and Johan Ovlinger. Aspectual collaborations – combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003. http://www.ccs.neu.edu/research/demeter/papers/ac-aspectj-hyperj.

[34] Karl J. Lieberherr, David H. Lorenz, and Johan Ovlinger. Aspectual collaborations: Combining modules and aspects. *The Computer Journal*, 46(5):542–565, September 2003.

[35] Karl J. Lieberherr, David H. Lorenz, and Pengcheng Wu. A Case for Statically Executable Advice: Checking the Law of Demeter With AspectJ. In Mehmet Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, pages 40–49, Boston MA, 2003. ACM Press.

[36] Karl J. Lieberherr, Jeffrey Palm, and Ravi Sundaram. Expressiveness and complexity of crosscut languages. In *Proceedings of the 4th workshop on Foundations of Aspect-Oriented Languages (FOAL 2005)*, March 2005.

[37] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Improving reusability and reducing development costs with adaptive software. Position paper for the Second International Workshop on Software Reusability in Lucca, Italy, March 24-26, 1993. Sponsored by IEEE Computer Society, December 1992.

[38] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[39] Karl J. Lieberherr and Ernst Specker. Complexity of partial satisfaction. *Journal of the Association for Computing Machinery*, 28(2):411–421, 1981.

[40] Karl J. Lieberherr and S. Vavasis. Analysis of polynomial approximation algorithms for constraint expressions. *Springer Verlag, Lecture Notes in Computer Science*, 145:187–197, 1983.

[41] Karl J. Lieberherr and Cun Xiao. Formal Foundations for Object-Oriented Data Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):462–478, June 1993.

[42] Karl J. Lieberherr and Cun Xiao. Minimizing dependency on class structures with adaptive programs. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 424–441, Kanazawa, Japan, November 1993. JSSST, Springer Verlag.

[43] Karl J. Lieberherr and Cun Xiao. Object-Oriented Software Evolution. *IEEE Transactions on Software Engineering*, 19(4):313–343, April 1993.

[44] Ling Liu, Roberto Zicari, Walter Hürsch, and Karl J. Lieberherr. Polymorphic reuse mechanisms for object-oriented database specifications. In *International Conference on Data Engineering*, pages 180–189, Houston, February 1994. IEEE.

[45] Cristina Isabel Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, Northeastern University, 1997. 274 pages.

[46] Cristina Videira Lopes. Adaptive parameter passing. In *2nd International Symposium on Object Technologies for Advanced Software*, pages 118–136, Kanazawa, Japan, March 1996. Springer-Verlag.

[47] Cristina Videira Lopes, Paul Dourish, David H. Lorenz, and Karl J. Lieberherr. Beyond AOP: Toward Naturalistic Programming. In Crocker and Jr. [5], pages 34–43. Onward! Track.

[48] David Mandelin, Lin Xu, Rastislav Bodik, and Doug Kimelman. Jungloid mining: Helping to navigate the api jungle. In *PLDI '05: Proceedings of the ACM SIGPLAN 2005 conference on Programming language design and implementation*, New York, NY, USA, 2005. ACM Press.

[49] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice-Hall International, 1988.

[50] M. Mezini and K.Ostermann. Conquering Aspects with Caesar. In Mehmet Aksit, editor, *Second International Conference on Aspect-Oriented Software Development*, pages 90–100, Boston MA, 2003. ACM Press.

[51] Mira Mezini and Karl J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. In C. Chambers, editor, *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*, number 10, pages 97–116, Vancouver, October 1998. ACM.

[52] Mira Mezini and Karl J. Lieberherr. Adaptive plug-and-play components for evolutionary software development. Technical Report NU-CCS-98-3, Northeastern University, April 1998. To appear in OOPSLA '98.

[53] Doug Orleans. *PROGRAMMING LANGUAGE SUPPORT FOR SEPARATION OF CONCERNS*. PhD thesis, Northeastern University, 2005.

[54] Doug Orleans and Karl J. Lieberherr. DJ: Dynamic Adaptive Programming in Java. In *Reflection 2001: Meta-level Architectures and Separation of Crosscutting Concerns*, Kyoto, Japan, September 2001. Springer Verlag. 8 pages.

[55] Klaus Ostermann, Mira Mezini, and Christoph Bockish. Expressive pointcuts for increased modularity. In *European Conference on Object-Oriented Programming*. To appear in ECOOP 2005. Currently available at `http://www.st.informatik.tu-darmstadt.de/public/StaffDetail.jsp?id=8`, 2005.

[56] Johan Ovlinger. *Combining Aspects and Modules*. PhD thesis, Northeastern University, 2004. 197 pages.

[57] Jens Palsberg, Boaz Patt-Shamir, and Karl J. Lieberherr. A new approach to compiling adaptive programs. In Hanne Riis Nielson, editor, *European Symposium on Programming*, pages 280–295, Linkoping, Sweden, April 1996. Springer Verlag. Lecture Notes in Computer Science 1058.

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

[59] David L. Parnas. On the criteria to be used in decomposing systems into modules. pages 411–427, 2002.

[60] Tobias Rho and Günter Kniesel. LogicAJ - Uniform Genericity for Aspect Languages. Technical Report IAI-TR-2004-4, Computer Science Department III, University of Bonn, December 2004.

[61] Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. XAspects Home Page. http://www.ccs.neu.edu/research/demeter/xaspects .

[62] Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. XAspects: An Extensible System for Domain Specific Aspect Languages. In Crocker and Jr. [5], pages 28–37. Special Track on Domain-Driven Development.

[63] Ignacio Silva-Lepe. An empirical method for identifying objects and their responsibilities in a procedural program. In *TOOLS Europe, Technology of Object-Oriented Languages and Systems*, pages 136–149, Versailles, France, 1993. Prentice-Hall.

[64] Ignacio Silva-Lepe. A model for migrating procedural programs into object-oriented programs. In *Technology of Object-Oriented Languages and Systems Pacific Conference*, pages 193–209, Melbourne, Australia, 1993. Prentice-Hall. Also available as technical report NU-CCS-93-12, Northeastern University.

[65] Ignacio Silva-Lepe, Walter Hürsch, and Greg Sullivan. A Report on Demeter/C++. *C++ Report*, February 1994.

[66] Kevin Sulivan, William G. Griswold, Yuanyuan Song, Yuanfang Cai, Macneil Shonle, Nisit Tewan, and Hridesh Rajan. On the criteria to be used in decomposing systems into aspects. In *European Software Engineering Conference and International Symposium on the Foundations of Software Engineering*, 2005.

[67] The AspectJ Team. AspectJ Development Tools, 2005. `http://www.eclipse.org/aspectj/`.

[68] Wim Vanderperren, Davy Suvee, Bart Verheecke, Maria Agustina Cibran, and Viviane Jonckers. Adaptive programming in JAsCo. In *AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development*, pages 75–86, New York, NY, USA, 2005. ACM Press.

[69] Robert J. Walker and Kevin Viggers. Implementing protocols via declarative event patterns. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 159–169, New York, NY, USA, 2004. ACM Press.

[70] Mitchell Wand. Understanding aspects (extended abstract). In *International Conference on Functional Programming*, August 2003.

[71] Pengcheng Wu and Karl Lieberherr. Shadow programming: Reasoning about programs using lexical join point information. In *GPCE '05: Proceedings of the 4th International Conference on Generative Programming and Component Engineering (to appear)*, 2005.