

Demeter Interfaces: Adaptive Programming without Surprises

Therapon Skotiniotis, Jeffrey Palm, and Karl Lieberherr

College of Computer & Information Science
Northeastern University, 360 Huntington Avenue
Boston, Massachusetts 02115 USA.
{skotthe, jpalm, lieber}@ccs.neu.edu

Abstract. Adaptive Programming (AP) allows for the separate definition of data structures and traversals with attached computations, performed during the traversal, that operate on these data structures. The loosely defined contexts, structure and computation, are composed according to a given traversal specification. Traversal specifications are defined against a graph-based model of the underlying data structure with the ability to abstract over graph node names, edges and subpaths. As such *certain* modifications to the data structure can be made without altering the programs overall behavior. The program *adapts*, or more precisely the computation can withstand, modifications to its underlying data structure. Currently AP systems, *i.e.* DAJ, provide no mechanisms to warn or even guard against modifications that *will* affect the meaning of a program. Programmers have to depend on thorough (or even exhaustive) testing in order to detect such modifications.

In this paper we present *Demeter Interfaces*, through which a more thorough design method of adaptive programs allows for more resilient software. Demeter Interfaces specify the expected structural properties of the underlying data structure that *must* hold in order for adaptive code to function correctly. Through an example we illustrate the usage and implementation of Demeter Interfaces. We further show how Demeter Interfaces result in better designs of adaptive programs, ease of adaptive code reuse and how they promote parallel development. We also discuss the applicability of Demeter Interfaces to XML technologies making our results relevant to large communities, such as the XQuery, XLink and XSLT communities.

1 Introduction

An adaptive program is written in terms of loosely coupled contexts, *i.e.*, data structure and behavior (computations) with a third definition succinctly binding the two contexts together. In DAJ [2], the most recent AP tool,¹ programs are defined in terms of their data structures and a set of computations that take place during the traversal of these data

¹ DemeterJ and DJ are alternative AP tools. The tools differ technically in the way computation is attached to traversals. DemeterJ [3, 4] generates Java code inside Java classes, DJ [5] deploys reflection and DAJ uses AspectJ [?,?] introductions. All three tools make use of the same core library (APLib [6, 7]) when calculating traversals.

structures. Structure and computation come together through *traversal specifications*. Traversal specifications are defined using a domain specific language that operates on a graph-based model of a program's data structure. Traversal specifications can abstract over graph node names, edges and subpaths thus allowing certain modifications to the underlying data structure that do not alter the program's overall behavior.

This adaptive nature of AP programs better lends itself towards iterative software development [8,9]. Programs are built in small iterations where each iteration adds a new small piece of behavior that operates on the underlying data structure. Typically modifications to the underlying data structure are necessary and often lead to code modifications from older iterations. The adaptive nature of AP systems assists in limiting, but not completely removing, such situations.

It is therefore important to identify the kinds of modifications to which an AP system will not "correctly" adapt and will yield undesirable program behavior. Identifying which modifications do not alter the meaning and intent of an AP system is not a trivial task. A naive approach would be to identify which parts of the data structure are explicitly referenced in traversal specifications and their attached computations. These data definitions have a dependency and any modifications to their name and/or structure can negatively affect the attached computation. These dependencies alone however do not suffice, since computations typically depend on implicit assumptions about the underlying data structure that a modification may invalidate.

As a simple example consider an application that collects information from a data structure that represents a bus route. The data structure consists of a list of `BusRoute` objects each one with a list of `Bus` objects as its data member. In turn, each `bus` maintains a list of `Person` objects as its member where each `Person` object holds the ticket price paid by each passenger. Calculating the total amount of ticket money collected due to the current bus passengers riding on a bus route requires a traversal to all `Person` objects, *i.e.*, "from `BusRoute` to `Person`", collecting the ticket price from each object along the way and adding the values together.

It is clear that the traversal specification depends on the names `BusRoute` and `Person`. More importantly, the traversal specification makes the implicit assumption that all `Person` objects reached through a `BusRoute` object are all bus passengers. Extending the data structure so that a `BusRoute` also holds a list of `BuStops` which themselves contain a list of waiting passengers does not invalidate the traversal specification, but calculates the wrong amount of ticket money collected. DAJ (as well as DemeterJ and DJ) offers no way to define and check for such assumptions. Programmers resort to extensive testing as the only mechanism for identifying this kinds of violations.

The problems due to modifications that alter the meaning of the program make iterative and parallel development difficult. As dependencies between computations and traversals arise it becomes harder to properly test and detect bugs in adaptive programs. At the same time, as AP programs become larger in size understandability of adaptive code decreases since programmers need to perform traversal calculations over large data structures by hand. Programmers have to both look at the traversal specification and the computation attached to it but also calculate from the concrete data structure the possible paths based on the traversal specification. Finally, since both traversal specifications

and their attached computations refer to concrete data structure names it is harder to reuse adaptive code as is.

In this paper we propose Demeter Interfaces (DIs) as a mechanism within DAJ that allows the definition of an *Interface Class Graph* (ICG), which provides an interface for the concrete data structure. A DI further specifies its relevant traversal files which consist of traversal specifications and *constraints*. Constraints define properties that both the ICG and the underlying data structure must satisfy. Computation is specified either as inter-type declarations (ala AspectJ) that introduce extra methods to classes, or as Visitors that are attached to traversals via adaptive methods. Visitors define methods that get to execute during the traversal of the data structure, *i.e.*, before and after specific nodes are reached.

We further extend the concrete data structure definition with an `implements` clause used to specify which DI(s) are implemented along with a name map between its concrete data members and the DI(s) data members. Finally we extend DAJ to statically verify the mapping provided and validate all constraints from the related Demeter Interfaces.

Demeter Interfaces hit a sweat spot between flexibility and safety. They restrict what AP can do but without going back to the old way of writing the Structural Recursion template manually [10]. They are safer because the adaptive program's intent is defined and used to check any future data type against it. As a side effect, adaptive programs become better documented and more understandable.

The remainder of this paper is structured as follows, Section 2 introduces DAJ Interfaces by presenting an example application implemented in plain DAJ and then with Demeter Interfaces. Section 3 discusses some of the implementation details and section 4 describes the design benefits enjoyed by adaptive programs that deploy DIs. Section 5 discusses the connection between DIs and XML technologies, section 6 presents related work and Section 7 presents future work. Section 8 concludes.

2 Demeter Interfaces

In this section we illustrate the usage of DIs and their advantages through an equation system example and the implementation of a semantic checker. We first provide a solution in DAJ [2, 11] which we also use to describe the DAJ system itself. We then iteratively extend the equation system, exposing the issues with the current DAJ implementation. We then show a solution using DIs and analyze the advantages over our initial implementation.

2.1 A simple equation system in Demeter.J

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 $x = 5; y = 9; z = x + y;$.

Adaptive programs in DAJ are defined through a Class Dictionary (*cd*), a set of Traversal Files (*trv*) and a mixture of Java and AspectJ code. Listing 1.1 and Listing 1.3 make up the DAJ code for the simple equation system along with the implementation

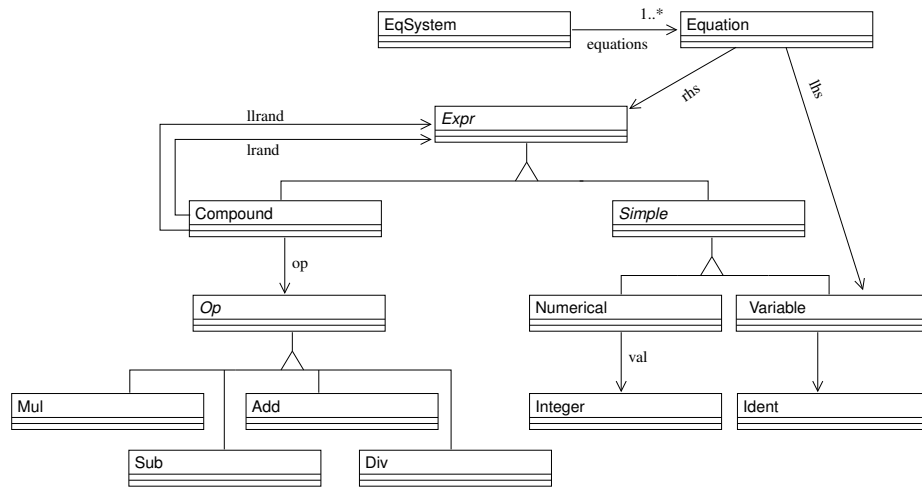


Fig. 1. The UML equivalent of Simple Equations Class Graph.

of the semantic checker which prints out two lists; defined and used variables. A cd file is a textual representation of the Object-Oriented structure of the program which specifies classes and their members. Figure 1 gives the UML representation of the class dictionary in Listing 1.1.

Listing 1.1. Class Dictionary for Simple Equations

```

EqSystem = <equations> BList(Equation).
Equation = <lhs> Variable "=" <rhs> Expr.
Expr : Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr)".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S } ")".
  
```

Each line of the class dictionary defines a class definition. An equal sign (“=”) defines a concrete class with the class name on the left hand side of the equals and the members of the class in the right hand side of the equals. Replacing the equal sign with a colon (“:”) defines an abstract class with its subclasses on the right of the colon. Names enclosed in “< >” define class member variable names, classes with no members are specified using an equal sign followed by a dot (A = .). The class dictionary further

defines a graph-based model of the program's structure, referred to as a *class graph* with each class (concrete or abstract) represented as a node and each member variable represented as an edge (Figure 1). Inheritance is also represented as an edge, as in UML class diagrams, but with the direction of the arrow is reversed to point to subclasses instead of the super class.

The system uses a class dictionary as a grammar definition, providing a language that can parse in sentences and create the appropriate object instances. Tokens in the class dictionary surrounded in quotes define the generated languages' syntax tokens (Listing 1.2). Parameterized classes are defined through the tilde (“~”) operator, *e.g.*, `BList(S)` defines a list enclosed in parentheses of one (or more) elements of type *S* each element separated by a semicolon.

Listing 1.2. An instance of a simple equations system given as input to DAJ

```
(x = 5;
y = (x - 2);
z = ((y-x) + (y+9)))
```

Traversal files are an extension to AspectJ's aspect definition which only allow inter-type declarations. DAJ extends AspectJ's `declare` statement to define strategies and traversals. Strategy declarations provide a name for the strategy (*e.g.*, `define` in Listing 1.3) and the traversal specification as a string. Traversal specifications can refer to class nodes by name and to class edges using the syntax `-> Source, Label, Target`. (*e.g.*, class *A* with member *c* of type *B* can be expressed as `->A, c, B`). In place of a class name or edge name the `*` pattern is used to match any name. In our simple equation system the strategy defined visits all `Variable` objects starting from an `EqSystem` object and bypassing any edge with the name `rhs` along the way.

Traversal declarations require a method signature and a strategy with a visitor name as an argument. The method signature provided to a traversal declaration gets introduced as a new public method to the source class of the traversal's strategy. We call these methods *adaptive methods*. DAJ automatically generates the method body that performs the necessary calls for traversing the object's structure according to the given strategy. At each such call the attached visitor implementation is advised, executing any applicable visitor method. Visitors in DAJ are Java classes where the one argument method names `before`, `after` and `return` hold a special meaning. During a traversal if an object's type matches the argument type of a `before` method then that method is called before traversing the object. After methods behave in a similar way with except that the method gets called after traversing the object. Return methods are used to return a value once the traversal is terminated. The return type of the `return` method must match the return type of the adaptive method that the visitor is attached to. In the simple equation system, before a `Variable` object is traversed the `CollectDef` visitor prints out the variable's name.

Listing 1.3. Additional traversal file (`SemanticChecker`), visitor class (`CollectDef`) and main driver class (`Main`) for the system of simple equations.

```
// SemanticChecker.trv aspect SemanticChecker /*[*]*/declare strategy : all: "from EqSystem
to *"; //declare traversal: void print(): all(PrintVisitor); /*[*]*/ declare strategy : defined:
```

```

    "from EqSystem bypassing -i *rhs,* to Variable"; declare traversal: void printDefined():
    defined(CollectDef);
    declare strategy : used : "from EqSystem bypassing -i *lhs,* to Variable"; declare traversal:
    void printUsed(): used(CollectDef);
    //
    // CollectDef.java
    // Visitor class CollectDef void before (Variable v) System.out.println("Found Ident : " +
    v.ident.toString());
    // Main Class import java.io.*; class Main public static void main(String args[]) try
    EqSystem eqSystem = EqSystem.parse(new File(args[0])); System.out.println("Defined are
    : "); eqSystem.printDefined(); System.out.println("Used are : "); eqSystem.printUsed();
    catch (Exception e) e.printStackTrace();

```

With the completed AP implementation of the semantic checker in place we can now evaluate our solution and verify the claims made, both in favor and against, AP. The principle behind AP [3] states

“A program should be designed so that the interface of objects can be changed within certain constraints without affecting the program *at all*.”

For the simple equation system example, modifying the system so that equations are now in prefix notation does not affect the program’s behavior. Doing so requires a single modification to the class dictionary,

```
Compound = ``('' <op> Op <lrand> Expr <rrand> Expr ``)''.
```

No other changes are needed to the traversal file or the visitor. The modification simply changed the order between the Op data member and the first Expr data member of the Compound class. This is not surprising since even in plain Java, switching the order of member definitions does not change a program’s behavior. Lets consider a more drastic extension, lets add exponent operations to our system but also impose precedence between operators. Listing 1.4 shows the complete class dictionary file, the definitions have been factored to accommodate for operator precedence and the ExpO class has been introduced to deal with exponents. Again, no other changes are need to either the traversal file or the visitor. The semantic checker still functions correctly.

Why is the semantic checker unaffected by these changes? In both cases the modifications to the cd file did not falsify the strategy (*i.e.*, there is still a path from source to the target) and it did not affect the way by which variables are defined and used in the equation system (*i.e.*, there is no other way of binding a variable to equations other than ``='’ and variables still have global scope). Any modification to the class dictionary that does not falsify the strategy and does not alter the assumptions about variable definition and usage within the equation system will not affect the semantic checker’s code.

However any alteration that either

- modifies class and/or class member variable names that are explicitly referenced by traversals and/or visitors,
- or, breaks an assumption about the system on which an adaptive code depends on (*e.g.*, adding a new variable binding construct to the equation system like `let` for local bindings).

will alter the program’s behavior.

Listing 1.4. Extended class graph accommodating exponents and operator precedence.

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> Variable "=" <rhs> Expr.
Expr : AddExp | SubExp | Term.
AddExp = Add Expr Term.
SubExp = Sub Expr Term.
Term : MulTerm | DivTerm | Expo.
MulTerm = Mul Term Expo.
DivTerm = Div Term Expo.
Expo : Raised | Factor.
Raised = "**" Expo Factor.
Factor : Simple | BExpr.
BExpr = "(" Expr ")".
Simple : Variable | Numerical.
Variable = Ident.
Numerical = <val> Integer.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ S {";" S}.
```

For example, altering the equations system to allow for function definitions with arguments causes no compile time error, but results in erroneous program behavior. This modification breaks two assumptions:

1. There is only one new `Variable` defined at each equation.
2. All variables have global scope and thus can be used anywhere.

The adaptive methods, as well as the visitor, depend on these assumptions. However these assumptions are not explicitly captured in AP programs. There is no tool support to stop such modifications. In fact naively extending the equation system to accommodate for functions parameters, as in Listing 1.5, will generate a valid AP program that will provide the wrong results for the semantic checker.

With larger AP programs, it becomes nearly impossible to find all these implicit assumptions and even harder to predict which modifications will cause erroneous behavior. Programmers have to rely on exhaustive testing in order to increase their confidence that the program still behaves according to its specification. This in turn limits the effectiveness of AP and its application in iterative development since modifications to the data structure due to an iteration can introduce bugs in parts of the code developed in previous iterations.

Listing 1.5. Class Graph for equations systems with functions of one argument.

```
EqSystem = <equations> BList(Equation).
Equation = <lhs> VarOrFunc "=" <rhs> Expr.
VarOrFunc : Variable | Function.
Function = "fun" <fname> Variable "(" <args> CList(Variable) ")".
Expr : FunCall | Simple | Compound .
```

```

Simple : Variable | Numerical.
Variable = Ident.
FunCall = <fname> Variable "(" <fargs> CList(Simple) ")".
Numerical = <val> Integer.
Compound = "(" <lrand> Expr <op> Op <rrand> Expr)".
Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".
CList(S) ~ S {"", " S}.

```

These dependencies impede parallel development and decrease productivity. Addressing these issues requires

- The ability to define the assumptions made by adaptive code about the underlying data structure,
- Tool support to allow the validation of these assumptions,
- Decrease the dependency on class and class member variable names,
- The modularization of only the relevant data structure information for each adaptive behavior instead of the whole class dictionary.

2.2 A simple equation system with Demeter Interfaces

A Demeter Interface resides between a class graph and the *implementation* of adaptive behavior, *i.e.*, adaptive methods and visitor implementations. A DI defines the interface class graph as well as a list of traversal file names each defining strategies, traversal and constraints for this DI. A traversal file also defines a list of visitor class names that are used in its traversal definitions.

Figure 2 shows the Demeter Interface for the simple equation system along with its traversal file and visitor implementation. A diagrammatical representation (in UML) of the DI's interface class graph is given on the right.

The ICG serves as an abstraction of any class graph implementation of the `ExprICG` DI in order for the strategies defined in `ExprICG` to be applicable. The interface class graph captures only the necessary structural information. For the purpose of a semantic checker the interface class graph needs to capture the notions of variable definition and variable usage. Any other information, *e.g.*, operator precedence etc., is irrelevant for the semantic checker. Any definition in the equation system is viewed by the semantic checker as either defining a new variable or using some variables in the definition body. This is depicted in Listing 2 which defines an `ESystem` as a set of definitions, each definition having a `def` defining an entity (`DThing`) and a `body` part that uses (possibly many) entities (`UThing`).

The header for the DI defines, using a `with` statement, the traversal file name that uses this DI, in this case `SemanticChecker`. The `SemanticChecker` traversal file defines in its header, using a `with` statement, the visitor required by the traversals defined in

```

// ExprICG.di di ExprICG with SemanticChecker
ESystem = List(Definition). Definition =
;def; DThing "=" ;body; Body. Body =
List(UThing). UThing = Thing. DThing =
Thing. Thing = . List(S) "(" S ")".

// SemanticChecker.trv File aspect
SemanticChecker with DVisitor declare
strategy: gdefinedIdents: "from ESystem
via DThing to Thing". declare strategy:
gusedIdents: "from ESystem via UThing
to Thing". declare strategy definedIdent:
"from Definition via DThing to Thing".

declare traversal: void printDefined():
gdefinedIdents(DVisitor);
declare traversal: void printUsed():
gusedIdents(DVisitor);
declare constraints: unique(definedIdent),
nonempty(gusedIdents),
nonempty(gdefinedIdents).

// DVisitor.java File
class DVisitor public void before(Thing t)
System.out.println(t.toString());

```

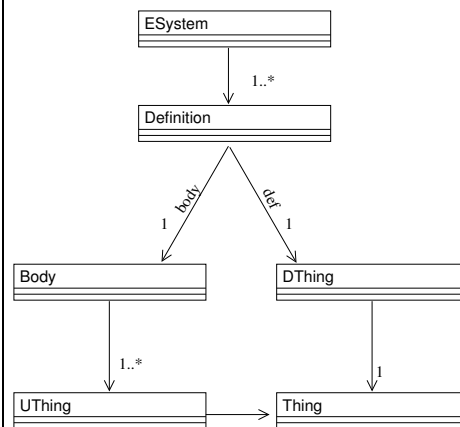


Fig. 2. The Demeter Interface for the simple equations system defines an interface class graph, the SemanticChecker traversal file defines strategies traversals and constraints. The UML diagram is equivalent to the interface class graph defined in ExprICG.

its body. The three traversals defined are, the first for collecting all defined entities in an `ESystem` (`gdefinedIdents`), the second for collecting all used entities in an `ESystem` (`gusedIdents`) and the last strategy collects all defined entities from a `Definition` object (`definedIdent`).

Following the strategy definitions, two adaptive methods are introduced into the `ESystem` class, `printDefined()` uses the `gdefinedIdents` strategy along with the `DVisitor` to collect all defined entities. In a similar manner `printUsed()` uses the `gusedIdents` strategy along with `DVisitor` to collect all used (referenced) entities. Finally the three constraints state the assumptions that must hold for the ICG and later for any class dictionary that is mapped to this ICG. Specifically, that each `Definition` has a unique path to a defined entity, *i.e.*, a `Definition` defines one and only one new variable. The remaining two constraints specify that both `gusedIdents` and `gdefinedIdents` should have at least one path satisfying their strategy specification. We have extended DAJ and these constraints are now validated statically.

A visitor implementation is a typical Java class containing one argument methods with the method name being either `before` or `after`. Execution of an adaptive method starts traversing the data structure according to the strategy provided with the traversal definition. Before traversing over an object, the visitor attached to this adaptive method is advised. If the type of the object to be traversed matches the argument type of a method, then the method is executed. As the method names imply, `before` methods are executed before traversing the object, `after` methods are executed after the object is traversed. Figure 3 gives an example implementation of the `ExprICG` Demeter Interface (on the right) along with a visitor implementation and a driver class (on the left). DAJ's class dictionaries are extended in two ways; a header is introduced allowing for an **implements** statement that specifies which DIs are being implemented and a mapping between the concrete class dictionary classes to the ICG's classes. The concrete class dictionary `InfixEQSystem` (Figure 3) provides a definition of its equation system and a mapping M between the classes in its class graph and all the classes in `ExprICG`'s interface class graph. The mapping definition can map class(es) to class(es), class member variable name(s) to class member variable name(s) and a class to the target of a strategy.

With the simple equation system implemented using Demeter Interfaces we now extend the system and verify that DIs assist the prevention of modifications that alter the program's behavior. We perform the same extensions as in Section 2.1 and show that in the situations where modifications did not affect the systems behavior are not affected by the incorporation of DIs. At the same time modifications that did result in erroneous program behavior before, result in compile time errors in the presence of DIs.

As a first evolution step we want to change from infix notation to prefix notation. This is a modification that does not alter the program's behavior even in the original DAJ solution. Moving to a prefix notation requires to change the definition of `Compound` in `InfixEQSystem` to

```
Compound = <op> Op <lrand> List(Expr) <rrend> List(Expr) .
```

<pre> import java.io.*; class Main { public static void main(String[] args){ try { InfixEQSystem ieqs = InfixEQSystem.parse(new File(args[0])); System.out.println("IDs in def:"); ieqs.printDefined(); System.out.println("IDs in use:"); ieqs.printUsed(); }catch(Exception e){ e.printStackTrace(); } } } </pre>	<pre> cd InfixEQSystem implements ExprICG { EquationSystem = <eqs> List(Equation). List(S) ~ "(" {S} ")". Equation = <lhs> Variable "=" <rhs> Expr. Expr : Simple Compound. Simple : Variable Numerical. Variable = Ident. Numerical = <v> Integer. Compound = <lrnd>List(Expr) <op> Op <rrnd>List(Expr). Op : Add Sub Mul Div. Add = "+". Sub = "-". Mul = "*". Div = "/". for ExprICG (use EquationSystem as ESystem, use Equation as Definition, use Expr as Body, use (->*,lhs,Variable) as DThing, use (->*,rhs,* to Variable) as UThing, use Variable as Thing) } </pre>
--	---

Fig. 3. `InfixEQSystem` defines a class graph and a mapping of the entities in the class graph to the interface class graph of `ExprICG`. The driver class `Main` uses the adaptive methods introduced by `ExprICG`.

Listing 1.6. Extending the class dictionary to accommodate function definitions using the ExprICG DI.

```

cd ParamEquations implements ExprICG{
  EqSystem = <equations> BList(Equation).
  Equation = <lhs> VarOrFunc "=" <rhs> Expr.
  VarOrFunc : Variable | Function.
  Function = "fun" <fname> Variable "(" <args> Variable)".
  Expr : FunCall | Simple | Compound .
  Simple : Variable | Numerical.
  Variable = Ident.
  FunCall = <fname> Variable "(" <fargs> Simple)".
  Numerical = <val> Integer.
  Compound = "(" <lrand> Expr <op> Op <rrand> Expr)".
  Op : Add | Sub | Mul | Div.
  Add = "+".
  Sub = "-".
  Mul = "*".
  Div = "/".
  BList(S) ~ "(" S {";" S} ")".

for ExprICG (
  use EquationSystem as ESystem,
  use Equation as Definition,
  use Expr as Body,
  use (->*,lhs,* to Variable) as DThing,
  use (->*,rhs,* to Variable) as UThing,
  use Variable as Thing
)
}

```

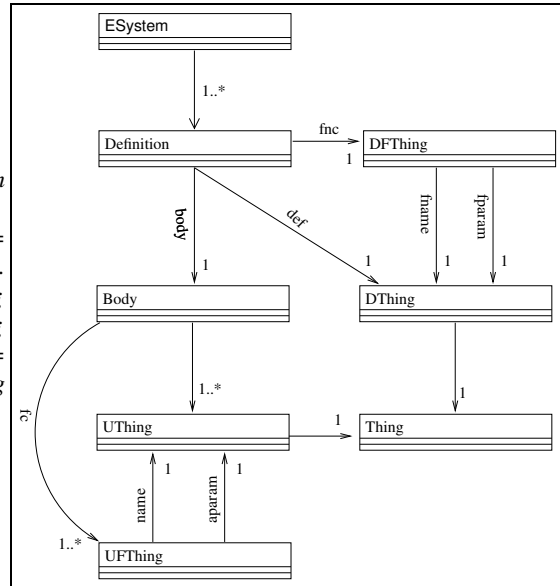
This change does not affect the Demeter Interface at all. We update the equation system class graph while keeping the original mapping M . All constraints of the DI are still satisfied after they are mapped into the actual interface class graph and the adaptive methods function correctly.

It is important to note that during this evolution step, only the DI and the concrete implementation of the interface class graph was needed. Under the assumption that the DI's constraints capture the semantic checker's intent, the static assurances provided by the tool because of the DI, suffice to show that the strategies pick the correct traversals and that the semantic checker still operates as expected. The Demeter Interface allows in this case for separate development and ease of evolution. The concrete class graph and its mapping can be maintained separately while adaptive code can be developed based on the publicly available DI. Alterations made to the concrete class graph do not need to be visible to adaptive code maintainers unless it affects the mapping to an implemented DI. This form of data hiding through the Demeter Interface also provides for easier maintainability and higher system modularity. As our next evolution step we extend the set of operators to include exponents and add operator precedence. Keeping the headers and mapping definition from `InfixEQSystem` and replacing the data structure definition by that of Listing 1.4 gives us a working AP system. The modifica-

```

// ParamExprICG.di File di ParamExprICG with
SemanticChecker
ESystem = List(Definition). Definition =
;def; DThing ;fnc; DFThing ;body; Body.
DThing = Thing. DFThing = ;fname;
DThing ;fparam; DThing. Body = ;fnc;
List(UFThing) List(UThing). UFThing =
;name; UThing ;aparam; UThing. UThing
= Thing. Thing =. List(S) "(" S ")".

```



```

// SemanticChecker.trv File import java.util.*;
aspect SemanticChecker with PVisitor declare strategy: definedIdents : "from ESystem to
DThing". declare strategy: usedIdents : "from ESystem to UThing". declare strategy:
dName : "from DFThing via -; *fparam,* to Thing". declare strategy: uName : "from
UFThing via -; *aparam,* to Thing". declare strategy: dFName : "from DFThing via -;
*fname,* to Thing". declare strategy: uFName : "from UFThing via -; *name,* to Thing".
declare traversal: LinkedList getDefined(): definedIdents(PVisitor); declare traversal: LinkedList
getUsed(): usedIdents(PVisitor); declare traversal:
LinkedList getDefName(): dName(PVisitor); declare traversal: LinkedList getUsedName():
uName(PVisitor); declare traversal: LinkedList getDefArg(): dName(PVisitor); declare
traversal: LinkedList getUsedArg(): uName(PVisitor);
declare constraints: nonempty(definedIdents), nonempty(usedIdents), unique(dName),
unique(uName), unique(dFName), unique(uFName)
//Inter-type definition public boolean ESystem.checkBindings(LinkedList l1, LinkedList l2) //
checks appropriate variable usage /* */ // PVisitor.java File class PVisitor public void
before(Thing t); public void before (DThing t); public void before (UThing t); public
LinkedList return(); /* */

```

Fig. 4. The evolved Demeter Interface and the UML representation of the extended Demeter Interface class graph.

tions to the data structure to accommodate for exponents and operator precedence do not invalidate any of the DI's constraints and in the resulting AP program behaves as expected.

In the next evolution step we want to add functions with one argument to the equation system. 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. Unlike definitions so far function parameters do not have global scope, their scope is local to the function definition. A naive approach would be to alter the class dictionary as in Listing 1.6.² Altering the data structure and only the mapping to `DThing` results in a compile time error. The reason is that the predicate `unique(definedIdent)` from `ExprICG` is no longer valid. The modification to allow functions with one parameter breaks one of the assumptions of the interface, in particular the fact that we can reach more than one variable through the left hand side of the equal sign. With one argument functions the meaning of what is defined and what is its scope has changed and these changes have to be reflected in the Demeter Interface.

It is important to note that for this evolution step the interface has to change (Figure 4). With a new interface class graph `ParamExprICG` we can abstractly reason about semantically checking systems with one argument functions. The two strategies `definedIdent` and `usedIdent`s are used to navigate to definitions and references of variable names, both function names as well as simple variables. The strategies `dName` and `uName` are then used to collect arguments (at function definition) and actual arguments (at function invocation) respectively. Similarly `dFName` and `uFName` collect function names at function definitions and function usage respectively. The traversal specifications use the strategies to collect `Thing` objects, while the implementation of method `checkBindings` is introduced into `ESystem` and it is used to check the correct usage of variable and function definitions. The inputs to this function are two lists where the first represents variable and function definition names at different scopes and the second represents names of variables and functions references at their corresponding scope.

Listing 1.7. Modifications to the concrete class dictionary to accommodate single argument functions.

```

cd ParamEquations implements ParamExprICG{
EqSystem = <equations> BList(Equation).
Equation = <lhs> VarOrFunc "=" <rhs> Expr.
VarOrFunc : Variable | Function.
Function = "fun" <fname> Variable "(" <args> Variable)".
Expr : FunCall | Simple | Compound .
Simple : Variable | Numerical.
Variable = Ident.
FunCall = <fname> Variable "(" <fargs> Simple ")".
Numerical = <val> Integer.
Compound = "(" <land> Expr <op> Op <rland> Expr)".

```

² To keep the example simple we do not allow the usage of function calls as arguments to other functions, *i.e.*, $f(f(3))$

```

Op : Add | Sub | Mul | Div.
Add = "+".
Sub = "-".
Mul = "*".
Div = "/".
BList(S) ~ "(" S {";" S} ")".

```

```

for ParamExprICG(
  use EqSystem as ESystem,
  use Equation as Definition,
  use (->,Equation,lhs,*) bypassing Function to Variable as (Definition,def,DThing)
  use (->,Equation,lhs,*) to Function as DFThing
  use (->,Function,fname,Variable) as (->,DFThing,fname,DThing)
  use (->,Function,args,Variable) as (->,DFThing,fparam,DThing)
  use (->,Equation,rhs,*) bypassing FunCall to Variable as UThing
  use Expr to FunCall as (->,Body,fc,UThing)
  use (->,FunCall,fname,Variable) as (->,UThing,name,UThing)
  use (->,FunCall,fargs,Variable) as (->,UThing,aparam,UThing)
  use Ident as Thing
)
}

```

Listing 1.7 shows the class graph that implements `ParamExprICG`. The class dictionary maps the edge `args` to the edge `fparam` and its source and target nodes accordingly. Also the `fargs` edge is mapped to to the edge `aparam` and `fname` is mapped to `name` with their source and target nodes mapped accordingly. `Function` is mapped to `DFThing` and `FunCall` to `UThing`. All reachable `Variable` objects via the `lhs` edge of `Equation` that `bypass` `Function` are mapped to `DThing`. In a similar way all `Variable` objects that can be reached from the `rhs` edge of `Equation` by `bypassing` `FunCall` are mapped to `UThing`. Figure 5 shows the visitor implementation and the driver class. The visitor interface defines the method `return` which is called by DAJ at the end of a traversal. The return value of the `return` method is also the return value of the traversal. In this evolution step, the Demeter Interface helped by disallowing a naive extension that would violate the intended behavior of the original Demeter Interface. The nature of the evolution required an extension of the interface and that resulted to changes in the driver class and a new concrete class dictionary. It is important to note how the Demeter Interface exposed the erroneous usage of the `ExprICG` interface for this evolution step and assisted in updating all the dependent components due to the definition of `ParamExprICG`.

3 Compiling Demeter Interfaces

The previous sections have covered most of the featured of Demeter Interfaces. In this subsection we provide a more detailed, informal, explanation of the current tool support for Demeter Interfaces in DAJ. We plan to introduce Demeter Interfaces to all Demeter Tools (DemeterJ and DJ) in the near future.

```

import java.util.LinkedList;

class PVisitor {
    LinkedList env;

    PVisitor(){
        this.env = new LinkedList();
    }
    public void before(Thing t) {
        env.add(t);
    }
    public void before(UFThing ud) {
        LinkedList rib = getUsedName();
        rib.addAll(getUsedArg());
        env.add(rib);
    }
    public void before(DFThing ud) {
        LinkedList rib = getDefName();
        rib.addAll(getDefArg());
        env.add(rib);
    }
}

public LinkedList return(){
    return env;
}
}

import java.io.*;

class Main {
    public static void main(String[] args){
        boolean codeOk ;

        PVisitor defV = new PVisitor();
        PVisitor useV = new PVisitor();
        ParamEquations pe = ParamEquations.
            parse(new File(args[0]));
        codeOk = pe.checkBindings(
            pe.getDefined(defV),
            pe.getUsed(useV));
        if (!codeOk)
            System.out.println(" Variables used before" +
                " they where defined");
    }
}

```

Fig. 5. Changes to the interface affect `Main`. The definition of `PVisitor` is used to check for the local parameter names in parametric equations.

The incorporation of Demeter Interfaces into DAJ has resulted into two new compilation steps for AP software. A Demeter interface along with its supporting code (traversal files and visitor implementations) is compiled through a separate phase. This phase typechecks traversal files and visitor implementations, *i.e.*, names in strategy definitions and in visitor methods are defined in the DI's ICG. Also, the return type for traversal specifications matches the return type of the visitors special `return` method. Finally, for this phase, the constraints defined inside traversal files are verified, and an archive of the necessary generated files is created.

The second compilation phase takes as input a concrete class dictionary and an archive generated as output from a compilation of all required DIs for this concrete class dictionary. Using the mapping in the concrete class dictionary, DI code is expanded for the specific concrete class graph. After expansion the DI constraints are verified and finally the complete AP application is generated as compiled Java class files.

3.1 Mapping Concrete Class Dictionary to the Interface Class Graph

A mapping can be thought of as a relation between **all** the classes in an ICG and the classes in the concrete class dictionary. Mapping directives allow mappings between:

- concrete class to an ICG class
- concrete edge (member variable name) to an ICG edge
- concrete edge (member variable name) to an ICG class
- strategy to an ICG class
- strategy to an ICG edge

Mapping an edge forces the source and target classes for that edge to get mapped as well. For example, **use** `->, A, b, C` **as** `-> D, e, F` will also map A to D and C to F. This brings about the situation where the same mapping for a class can be specified more than once. This is not an issue as long as the DI predicates are still valid under the name mapping. If the `*` pattern is used as the target of a strategy in a mapping then all possible targets in the concrete class dictionary get mapped to the same ICG entity.

Since a concrete class graph can implement more than one DI, different DIs may map the same concrete class to different ICG classes. DAJ internally uses the name of the DI and the name of the ICG class for the mapping to resolve name classes and name ambiguities, these issues come into play at strategy expansion. In the case where a strategy is used to map to an ICG class or edge the mapping takes into account the strategy itself and not just the target class or edge. The case where an edge is mapped to a class is treated as in the case of a simple strategy being mapped to a class.

3.2 Constraints

Constraints may be placed on strategies of a DI that further restrict the implementing class graphs. These are specified declaratively in the DI in a separate section as shown in Figure 2. Constraints are evaluated on *Traversal Graphs* and not the whole class graph. Traversal graphs are a specific view of a strategy under a specific class graph. That is, irrelevant nodes and edges that cannot help in satisfying the strategy *s* under

a class graph g are removed. The AP Library represents traversal graphs as objects parameterized by a strategy and a class graph.

Currently the following primitive constraints for strategies s and t are provided in DAJ:

- **unique**(s): There must be only one path in the traversal graph representing s ; hence this path can statically be determined. The traversal graph for s is *unique* if there are no loops and all nodes have exactly one outgoing construction edge.³
- **nonempty**(s): There exists at least one path in the traversal graph representing s .
- **subset**(s,t): All the traversals defined by s are a subset of those in t . Put another way, if a traversal is defined by s , then it is also defined by t .
- **multiple**(s): There exist more than one path in the traversal graph representing s .

All primitives can be combined using common logical operators such as and ($\&\&$), or ($\|\|$), and negation $!$. With this ability one could, for example, define equivalence of two strategies s and t as:

$$\mathbf{equiv}(s,t) = \mathbf{subset}(s,t) \ \&\& \ \mathbf{subset}(t,s).$$

The tool implements these primitives through functionality of the current AP Library [7]. **nonempty** can be implemented in the current release, while **subset** and **unique** are implemented through calls to the new interface `AlgorithmsI`⁴:

```
interface AlgorithmsI {
    Descriptive.Boolean isSubset(TraversalGraph t1,
                                TraversalGraph t2);
    Descriptive.Boolean isUnique(TraversalGraph t);
}
```

3.3 Strategy Expansion

At strategy expansion DAJ uses the mapping provided with the concrete class graph and all the related files (the archive generated by the DI's compilation) and regenerates the files based on the name mapping.

DAJ goes through the DI's traversal files and visitor implementations and systematically replaces strategy definitions and visitor method argument types according to the mapping provided by the concrete class dictionary. The translation is straight forward for when a class maps to a class. In the case where an edge is mapped to a class or a strategy is mapped to a class then this class cannot be an argument to a visitor method. Currently in DAJ visitor methods cannot deal with edges and so any mapping that takes an ICG class to an edge or strategy causes a compile time error. Finally, strategies that are part of a mapping directive may have to be altered before they are replaced inside the DI's strategy definitions. The last "to" directive of the strategy given in the mapping is

³ *Construction edges* are edges in a class graph that represent class members. *Alteration edges* refer to edges that represent inheritance links.

⁴ `Descriptive.Boolean` is a utility class that contains a `boolean` value and descriptive reason *why* that value was returned.

replaced by a “via” directive if it is replacing any other segment of a strategy other than the target. If the strategy from a mapping definition is replacing the target class of a DI’s strategy then the last “to” directive of the DI’s strategy is replaced by a “via” directive. In this way, the rewrite ensures that the resulting strategy is syntactically valid.

4 Modularity and Demeter Interfaces

The introduction of Demeter Interfaces to AP development assists in designing, maintaining and understanding adaptive programs. The ideas behind DIs and their usage has revealed several design benefits.

During the design process of adaptive programs developers would first design a minimal class dictionary. Then iteratively both adaptive code and the class dictionary itself are developed with repeated testing to verify the behavior of adaptive code. Modifications to both the class dictionary as well as the adaptive code (the traversal specification and/or the visitor attached) were necessary. As programs become larger in size distinguishing which parts of the class dictionary are involved in the different adaptive methods becomes difficult. Furthermore, modifications to class names in the class dictionary cause changes to traversal strategies and/or visitor methods due to the lack of abstraction over class names. For example, in the development of CONA [12, 13] a Design by Contract (DbC) extension to Java and AspectJ, the class dictionary is the whole Java and AspectJ language syntax. Understanding the dependencies between adaptive code and the class dictionary becomes a laborious and error prone process.

DI provide solutions to both of these problems. The icg provides an abstraction of the concrete class graph while the adaptive methods, traversal strategies and visitor interfaces localize all the information necessary for understanding the dependencies between adaptive code and the rest of the program. The mapping mechanism removes the tight dependence on naming conventions by providing an automatic renaming mechanism. The usage of DI allows for more modular AP designs.

To support our claim of modularity for DIs we borrow the definition for modular implementations as proposed by Kiczales and Mezini [14]

- 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.

DIs are textually local with traversal strategies and visitors specifying exactly how adaptive methods interact with the rest of the system. DIs are an abstraction of the implementation both of the class dictionary, through the ICG, but also through the visitor interfaces that are part of the DI’s definition. The extensions made to the DemeterJ system provide automatic mechanisms that both check that modules satisfy their own

interfaces as well as the interfaces of other modules. Composition of DI automatically managed by DAJ and configuration of the composition can be controlled via the implements keyword and the mapping specification.

5 Demeter Interfaces and XPath

Ideas in AP can be found in other technologies where the separation between navigation code and computation is necessary. According to the abstractions that traversal specifications allow, the problems of surprise behavior are present in these systems as well. XML and XPath queries are technologies widely used today that share similar issues with AP. Specifically one can think of DTD as class dictionaries and XPath expression as traversal strategies. The problems of surprise behavior are prominent in these technologies as well since modifications to the XML document might break assumptions that the XPath query depends upon. Consider the following DTD

```
<?xml version="1.0"?>
<!ELEMENT busRoute (bus*)>
<!ELEMENT bus (person*)>
<!ELEMENT person EMPTY>
<!ATTLIST bus number CDATA #REQUIRED>
<!ATTLIST person
  name CDATA #REQUIRED
  ticketprice CDATA #IMPLIED>
```

That defines `busRoute` containing list of buses. Each `bus` contains a list of `person` which in turn contains a `name` for the person and a `ticketprice` for the bus fare. Consider the following XPath query

```
var nodes=xmlDoc.selectNodes("//person")
```

that collects all `person` elements from a `busRoute`. We can think of a simple JavaScript that will iterate through `nodes` and calculate the total amount of money received by the current passengers riding the bus.

Thinking of the correspondence DTD to class dictionary and XPath query to strategy it is straightforward to create a corresponding DAJ program. In fact, the two systems are so alike in this respect that they also share the same problems when it comes to modifications of their underlying data structure.

Leaving the JavaScript code and the XPath query the same we can extend the DTD to accommodate for villages with bus stops along the bus route.

```
<?xml version="1.0"?>
<!ELEMENT busRoute (bus*,village*)>
<!ELEMENT bus (person*)>
<!ELEMENT village (busStop*)>
<!ELEMENT busStop (person*)>
<!ELEMENT person EMPTY>
<!ATTLIST bus number CDATA #REQUIRED>
<!ATTLIST person
```

```
name CDATA #REQUIRED
ticketprice CDATA #IMPLIED>
```

The resulting amount this time is not the total of all passengers riding the bus, but instead the total amount for all passengers, both riding and waiting at `busStops`. Similar problems are found in other XML technologies that use XPath like mechanisms, such as XQuery and XQuery, to select elements from a graph like structure.

Ideas from Demeter Interfaces can help to stop this kind of situations. Just like any XML document can define the DTD to which it confronts to, DTDs can define the XPath interfaces that they support and a mapping between the DTD elements and the XPath interfaces interface elements. For instance, if the current total of all passengers riding the bus is to be supported by the DTD representing bus routes then it should make available an interface with XPath queries and constraints on these queries. The constraints are the guarantees provided to programmers upon which they can base their code that will operate on valid XML documents for the specific DTD. At the same time, the mapping between the interface and the DTD itself allows for changes to the DTD to both names of entities as well as structure within the bounds of the constraints without imposing modifications to client code.

The usage of Demeter Interfaces also provides a clear distinct separation of responsibilities. In the case where an modification to the DTD breaks one of the XPath constraints then the blame lies with the DTD maintainer for breaking an interface that the DTD claims to implement. Taking the argument in the other direction, any code that depends on DTD properties that are not explicitly exported through an XPath interface, the developer for this code is to blame when modifications to the DTD alter the codes behavior.

6 Related Work

Ovlinger and Wand [10] propose a domain specific language as a means to specify recursive traversals of object structures used with the visitor pattern [15]. The domain specific language further allows for intermediate results from subtraversals allowing for a more functional style visitor definitions. The explicit full definition of the recursive data structure provides an interface between visitors and the underlying data structure. This approach enforces that each object in a traversal is explicitly defined allowing no room for adaptiveness.

Modularity issues in AOSD [14, 16, 17] have received great attention recently. Kiczales and Mezini [14] 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 [16] 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, Bill Griswold et. al. [17] advocate an XPI (crosscutting program interface) as a means to achieve separate development and explicit dependencies between implementations of crosscutting concerns and base code. DIs can be viewed as a specialization of XPI for Demeter Systems. A more recent paper [18] by the same authors, demonstrated (partially) mechanized checking of XPIs through the usage of aspects which check for the appropriate interface constraints.

Kiczales and Mezini in [19] 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.

7 Future Work

Work is currently under way for allowing visitor methods to advice edges removing the limitation that classes mapped to edges cannot be advised by visitors.⁵ The mapping definition can become long and difficult at times. A GUI tool that will help visualize ICG and class dictionaries as graphs will assist developers. Also a naive inferencing engine that can match mappings by structure, *i.e.*, mapping a class will automatically map its edges by position (as they appear in the class dictionary and the ICG) is already under development.

Another extension would be the provision of contracts on adaptive methods. Collaborations of adaptive methods exchange data and certain assumptions are being made that are not explicitly captured by the current system. The addition of pre- and post-conditions would assist in both defining and validating these assumptions. The composition of Demeter Interfaces to provide new Demeter Interfaces is also an interesting future research direction. Also the interactions between DI in an AP program still remains an open issue.

A long term goal is the extension of ideas behind Demeter Interfaces to more general purpose AOP languages. Specifically exposing a graph based model for the control flow of programs upon which *control interfaces* of their expected behavior can be defined. Each program module, *e.g.*, Java class or Java package, will be accompanied by a control interface that will specify an abstraction of the call graph within that module along with constraints. Aspects that are to be attached to this module will have to do so based on the module's interface. A composition of two modules will generate a composition of their corresponding control interfaces. Aspect developers and class developers will first agree on the control interface of their modules and the assumptions that it

⁵ For the latest release of DAJ with support for Demeter Interfaces visit DAJ's Beta Release web page [20].

must uphold. The implementation of the module and the aspects around it will then be checked against the agreed upon control interface. Similar ideas for interfaces between aspects and base code have been recently discussed [16, 17, 19].

8 Conclusions

We introduce Demeter Interfaces as an extension to Adaptive Programming. Demeter interfaces encapsulate all the information and dependencies between the adaptive code and the underlying data structure. Through the definition of an interface class graph and a set of graph constraints Demeter Interfaces impose restrictions on any concrete data structure to which adaptive code will be attached. These restrictions are enforced at compile time disallowing modifications to the underlying data structure that would otherwise provide incorrect program results. With Demeter interfaces in place we have shown how modularity as well as understandability of adaptive programs increases dramatically leading to better program design and promotes parallel development.

References

1. The Demeter Group: The DAJ website. <http://www.ccs.neu.edu/research/demeter/DAJ> (2005)
2. Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston (1996) 616 pages, ISBN 0-534-94602-X.
3. Demeter Research Group: Online Material on Adaptive Programming and Demeter. In: <http://www.ccs.neu.edu/research/demeter/>. Northeastern University (1989-2001)
4. Orleans, D., Lieberherr, K.J.: DJ: Dynamic Adaptive Programming in Java. Technical Report NU-CCS-2001-02, College of Computer Science, Northeastern University, Boston, MA (2001) <http://www.ccs.neu.edu/research/demeter/biblio/DJ-reflection.html>.
5. AspectJ Team: AspectJ home page. <http://aspectj.org> (2005)
6. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In Knudsen, J., ed.: European Conference on Object-Oriented Programming, Budapest, Springer Verlag (2001)
7. Lieberherr, K., Patt-Shamir, B., Orleans, D.: Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.* **26**(2) (2004) 370–412
8. Doug Orleans and Karl J. Lieberherr: AP Library: The Core Algorithms of AP: Home page. <http://www.ccs.neu.edu/research/demeter/AP-Library/> (1999)
9. Lieberherr, K.J., Orleans, D.: Preventive program maintenance in Demeter/Java (research demonstration). In: International Conference on Software Engineering, Boston, MA, ACM Press (1997) 604–605
10. Lieberherr, K.J., Riel, A.J.: Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming* **1**(3) (1988) 8–22 A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.
11. Ovlinger, J., Wand, M.: A language for specifying recursive traversals of object structures. In: OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (1999) 70–81

12. Sung, J.: Aspectual Concepts. Technical Report NU-CCS-02-06, Northeastern University (2002) Master's Thesis, <http://www.ccs.neu.edu/home/lieber/theses-index.html>.
13. Skotiniotis, T., Lorenz, D.: Conaj: Generating contracts as aspects. Technical Report NU-CCIS-04-03, College of Computer and Information Science, Northeastern University (2004)
14. Skotiniotis, T., Lorenz, D.H.: Cona: aspects for contracts and contracts for aspects. In: OOP-SLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, New York, NY, USA, ACM Press (2004) 196–197
15. Kiczales, G., Mezini, M.: Aspect-oriented programming and modular reasoning. In: ICSE '05: Proceedings of the 27th International Conference on Software Engineering, New York, NY, USA, ACM Press (2005) 49–58
16. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995)
17. Aldrich, J.: Open Modules: modular reasoning about advice. In: European Conference on Object-Oriented Programming. (2005)
18. Sullivan, K., Griswold, W.G., Song, Y., Cai, Y., Shonle, M., Tewari, N., Rajan, H.: 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)
19. Sullivan, K., Griswold, W.G., Song, Y., Shonle, M., Tewari, N., Cai, Y., Rajan, H.: Modular software design and crosscutting interfaces. In: IEEE Software, Special Issue on Aspect Oriented Programming. (To Appear, Jan/Feb 2006)
20. Kiczales, G., Mezini, M.: Separation of concerns with procedures, annotations, advice and pointcuts. In: European Conference on Object-Oriented Programming. (2005)
21. The Demeter Group: The DAJ beta website. <http://www.ccs.neu.edu/home/skotthe/daj> (2005)