Connectors for Reusable Aspects

Karl Lieberherr [1] David Lorenz [1] Mira Mezini [1,2]

[1] College of Computer Science
Northeastern University
Boston, Massachusetts 02115-5000
Email: {lieber,lorenz,mira}@ccs.neu.edu

[2] University of Siegen, D-57068, Siegen, Germany,
E-mail: mira@informatik.uni-siegen.de

October 30, 1999

Abstract

Aspect-oriented programming (AOP) controls tangling of concerns by isolating aspects that cross-cut each other into building blocks. Component-based programming (CBP) supports software development by isolating reusable building blocks that can be assembled and connected in many different ways. We show how AOP and CBP can be integrated by introducing a new component construct for programming class collaborations, called *aspectual component*. Aspectual components extend adaptive plug-and-play components (AP&P) with a modification interface that turns them into an effective tool for AOP. A key ingredient of aspectual components is that they are written in terms of a generic data model, called a participant graph, which is later mapped into a data model. We introduce a new property of this map, called *instance-refinement*, to ensure the proper deployment of components. We show how aspectual components can be implemented in Java, and demonstrate that aspectual components improve the AspectJ language for AOP from Xerox PARC.

1 Introduction

Separation of concerns and *modularity* are at the heart of the programming process. Concerns are abstract conceptual units, *aspects*, in which we decompose a given problem. Modules are software units, pieces of code, the building-blocks that help organize software. In the C programming language, modules are associated with files. In Ada, modules are packages. In *Object-Oriented Programming Languages* (OOPL), the main modules are objects and classes. However, procedures, functions, subroutines, and alike, which are found in almost any language, are also modules, since they too have an independent and well-defined interface. This definition of modules is well expressed in the *Aspect-Oriented Programming* (AOP) terminology: modules are called *generalized routines*.

Selecting a programming language is choosing the constructs for capturing abstractions into discrete modules. Modular programming is conducted by expressing the units of concern in the units of modularity. Hence, separation of concerns and modular programming are inherently coupled. AOP targets at uncoupling the two when the units of concern (*aspects*) *cross-cut* the units of modularity (*generalized routines*). The key point of AOP is that existing languages do not properly support modularity in the presence of *aspects* that do not "fit" in the modular structure.

AOP permits you to express each aspect separately, in terms of its own modular structure, possibly with some representation of the join points and have some way to merge their different modular structures based on resolving their join points. However, the modular break downs of different aspects that mutually affect each other do not necessarily conform to each other. Aspects, come with their own modular structures which cross-cut each other. In this paper we illustrate how explicit *connectors* can help overcome this difficulty, and furthermore, make aspect more *reusable.*

We take a broad view considering an aspect to be a *slice* of *single-* or *multi-*party functionality corresponding to a well-defined unit of concern. *Slice* indicates that an aspect is not necessarily self-contained. An aspect is not just a slice of *single-party* functionality, but, moreover, a slice of *multi-party* functionality that involves a protocol between several participants. An aspect is, in general, mutually affected by other aspects (at multiple join points spread throughout the other aspects), and the complete system is, in general, a collection of slices of functionality weaved by resolving their mutual join points.

Consider the Tic-Tac-Toe example in Figure **??** taken from the AspectJ tutorial. The example shows how to separate the different concerns of implementing the Observer pattern in the context of a board game. The game shown is Tic-Tac-Toe. Two (or more players), each modify the board-game and need to be notified whenever some of the other players has changed the board. This example is a powerful illustration of the AOP technique. The "big picture" was broken into smaller "pieces of concern", which improve the programming style. In this paper, we take the articulation of aspects one step further and also make the "pieces" *reusable.*

Unfortunately, a few symptoms limit the reusability of the aspects in the code fragment shown in Figure **??**. First, the name of the aspects introduce an undesired coupling between the name-spaces of the aspects and the main code. The prefix "`TTT`" of the aspect name suggests it must be applied to the Tic-Tac-Toa whereas the Observer pattern is a general pattern, and one may think this aspect can be weaved in a different context too. Second, the the naming "`TicTacToe`" in the aspect code and the names of methods. It is a known programming problem, namely, taking too much implementation detail into the abstraction. In this case, we see it in the name-space and in the implementation of `update`. As a result, we get limited reuse of the abstraction, aspects in our case.

What is missing in aspects are *plugs* that would allow not to assume any application specific knowledge prior to the actual connecting. The main contribution of this paper is in providing these kind of connectors for aspects. We show that these connectors promote reuse of aspects, thus making aspects more reusable.

1.1 The Role of Connectors

A connector, as its name indicates, connects components. The simplest connector is just *glue* that keeps components together. Sometimes a connector is a *wire* stretched between two components, establishing a pair-relationship [1] for a channel of communication. In other cases, a connector is a *bus*, allowing the transfer of information from one connected component to any other. By shifting more and more functionality away from the components and into the connectors, a complex connector becomes more and more like *middle-ware*, e.g., a connector might be a *message routing broadcasting and filtering device* [3].

In a general sense, a connector *bridges the gap* between components. A connector could be an *interaction* relationship [1], for example, an adapter between two Java beans. A connector may also be an *implementation* relationship [1], e.g., an inheritance-like relationship could be viewed as an implementation connector. In *Adaptive Plug&Play* (AP&P) component [21], a connector is an adapter, but one that performs a mapping of multiple roles in synchrony. Connectors of AP&P component map from roles to roles or from roles to classes. The connector has a role of its own, and the mapping is used for bridging the gap between the generic role behavior and the specific desired behavior.

Viewing connectors as edges and components as the nodes they connect, connectors can be characterized by the type of components they connect. If you have n kinds of components, then there are 2^n kinds of connectors (or 2^{n-1} non-directional connections.) IBM's VisualAge [10], for example, determines which connector to use according to whether an event is connected to an event or to a property. The former is directional, i.e., it matters for event propagation who sends and who receives the event, whereas in the latter it does not matter in which direction you pull the line because only one direction makes sense.

Mixing regular and AP&P component, for example, we would have two kinds: *generic* and *specific* components. We would then have four kinds of connectors. The role of the connector would vary depending on the kind of the components it connects. A generic-to-specific connector would be responsible for performing a mapping. A generic-to-generic connector would need to unify the abstractions of two AP&P component;

[1] We use the term *pair* and not *binary* to avoid possible confusion with other meanings of the word *binary.*

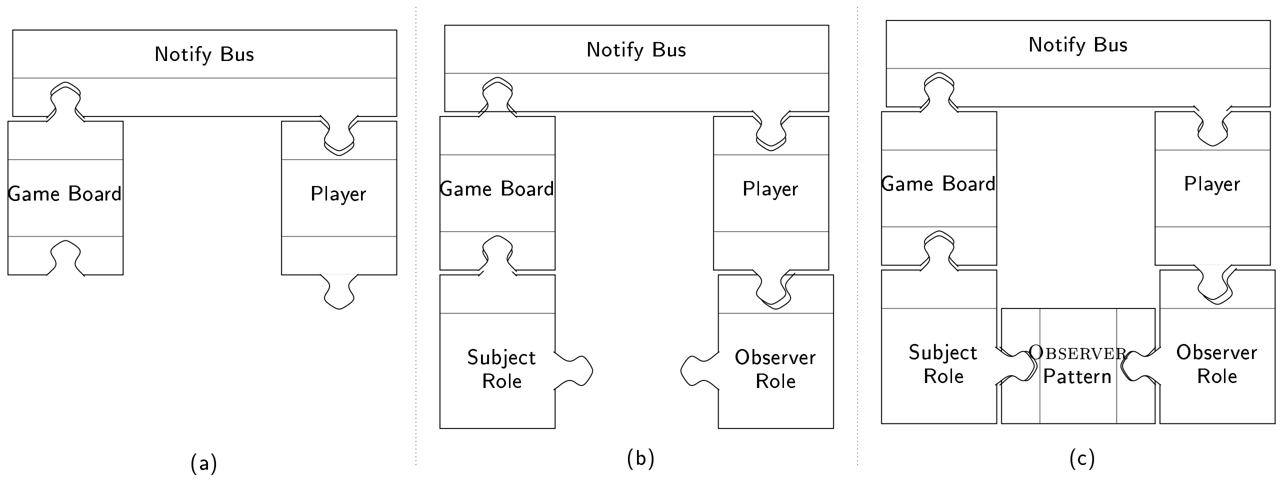

Figure 1: Observer pattern connectors: (a) interaction (b) participation (c) collaboration

and a specific-to-specific connector would simply be an interaction connection establishing a communication channel between the two.

The difficulty arises when all three kinds are used simultaneously. To illustrate this, consider the component-based approach, shown in Fig. 1, for implementing an Observer [6] pattern. In a board game, each player is an observer who needs to be notified whenever the board game is changed by another player. The board and the player are components. Connecting the board and the player via a notification bus, as in Fig. 1(a), is an example of connectors between specific components. The connectors in this case represent communication channels between the board and the player: whenever the board changes it *updates* the player about the change.

The player and the board need to follow their respective parts in implementing the standard Observer pattern. Taking an AOP approach [11, 16], instead of hard coding the subject and observer roles into the two participants, the roles may be obtained from generic components representing those *aspects* respectively. Fig. 1(b) adds connectors between a generic- and a specific-component. Each connector in this case bridges the specific-generic gap. The connector does not necessarily introduce a communication channel. Instead, it might be shifting code at assembly time only.

However, it is not enough that a board assumes the role of a subject and that the player assumes the role of an observer. In order for the construction to be *safe*, i.e., to work properly, the board and the player must be the subject and the observer of the same game, respectively, i.e., pass one another the same type of state. Fig. 1(c) shows a third kind of connector, which connects the two generic components. Connecting these two components represents the implicit constraints that need to be preserved when implementing the Observer pattern. By completing the connection, a violation of the constraints is eliminated. Whether or not this connection should also include the *interaction* notification-channel, and thus render the notification-bus component redundant, is an open question that this research would try to answer. We tend to keep them both. Connecting the game board to the subject role gives it the functionality of registering observers and notifying them when a change occurs. Connecting the player to the observer role gives it the functionality of being updateable. But the actual registration and notification is done by the notify-bus at runtime.

Outline The outline of the paper is as follows. In Sec. 3 we present the aspectual component model, and in Sec. 4 we illustrate its features by an example suite. Section 5 presents an implementation strategy for the model. Related works are described in Sec. 6. Sec. 7 concludes the paper.

2 Example

An example of a slice of a single-party aspect is tracing a program's execution in order to spot recurring patterns of method invocations or object access [?]. A description of this aspect in informal English would be: *For any data type in the application with some read access operation, say the class DataToAccess with the operation AnyType readOp(), and before any invocation of this operation on an instance of DataToAccess, say dataInstance, display the message: "Read access on <string representation of dataInstance>".* This functionality is single-party since it involves a single abstraction, DataToAccess. It is a slice, because it makes sense only in combination with a system's base behavior: it needs to be injected in the right places (join points) of a system's base behavior. DataToAccess and AnyType readOp() are just a place holders for actual classes and methods in a system's basic functionality (which is itself just another aspect). An example of a multi-party aspect would be a publisher-subscriber protocol [6]. It is slice, because the protocol makes sense only when injected (weaved) into certain points in a system's basic functionality.

Now, assume that the basic functionality is a graphical editor. Given the modular structure of object-oriented languages, each aspect alone is expressible in it. The graphical editor application will be decomposed into classes such as, Editor, Point, Line etc., with their corresponding functional and structural relationships. The same can be said for the tracing aspect. Its decomposition consists in a single class, e.g., our DataToAccess with a single operation AnyType readOp(), implemented as a display operation. The publisher subscriber protocol can also be expressed in a clean way in terms of two classes, Publisher and Subscriber. Both classes are related in a one-to-many structural relationship: a publisher has a list of associated subscribers. The publisher has two operations attach and detach for adding and removing subscribers from its list. It has an operation changeOp() which is implemented to invoke an update() operation on all subscribers. Thus, any aspect can be itself expressed in the modular structure. What is missing here, however, is a place (construct) where to express the weaving information, i.e., how the aspect participants and application classes are merged together. For instance, there is no explicit place where to express that all operations in Point whose name matches the get* pattern should be traced. That is to say, DataToAccess has to be merged with Point, whereby the implementation of readOp() should be merged before the implementation of all operations matching get*.

In the absence of a construct that supports the separate definition of aspects and a clean definition of their join points, one needs to spread *fragments of one aspect* around the module structure of another aspect and manually weave them within individual modules, resulting in tangled code. For instance, one way to achieve method tracing information is by source code instrumentation, adding to the program additional lines of code that log its execution. Clearly, this results in code tangling of the worst kind. The problem is twofold. On one hand the application code becomes more difficult to understand, modify and reuse. On the other hand, aspects lose their existence as a unit, making them extremely difficult to reuse, modify and be plugged in and out on demand.

These observations reinforce what *Aspect-Oriented Programming* (AOP) says all along: language constructs are needed to capture aspects that classes cannot (aspects are beyond classes). The difficulty, however, is in defining *what* such a language construct for object-oriented languages should look like. The only construct we have seen so far is the aspect construct of the AspectJ language [27]. The problem with the AspectJ proposal is that while doing a good job in localizing aspects in well-defined units, it does not provide a clean solution with respect to expressing join points. The approach is based on *lexical* cross-cutting. This means that the join points of an aspect to other aspects consist of names that appear in the implementation of the other aspects. For instance, in the case of tracing method invocations, the names of classes and operations that are affected are explicitly mentioned in (are part of) the definition of the tracing aspect. As a result, the aspect's applicability is restricted to the single *context of use* hard-wired in it. This damages the modularity of aspect definitions themselves. For illustration, two aspects described in the AspectJ tutorial [27] are given below.

```
aspect ShowAccess {
  static before Point.get,
                Point.getX,
                Point.getY {
    System.out.println("R");
  }
}

aspect TTTDisplayProtocol {
  static new Vector TicTacToe.observers = new Vector();
  static new TicTacToe TTTObserver.game;
  static new void TicTacToe.attach(TTTObserver obs) {
    observers.addElement(obs);
  }
  static new void TicTacToe.detach(TTTObserver obs) {
    observers.removeElement(obs);
  }
  static new void TTTObserver.update() {
    board.update(game); status.update(game);
  }
  static new void BoardDisplay.update(TicTacToe game),
                  StatusDisplay.update(TicTacToe game) {
    theGame = game; repaint();
  }
  // all methods that change state
  static after TicTacToe.startGame, TicTacToe.newPlayer,
               TicTacToe.putMark, TicTacToe.endGame {
    for (int i = 0; i < observers.size(); i++)
      ((Observer)observers.elementAt(i)).update();
  }
}
```

Both aspects above are written for particular applications. The first one for an application that has a class called Point. The publisher-subscriber aspect is written for a game application that has a class for the TicTacToe game and some other classes modeling the BoardDisplay of the game. This restriction is not justified, since both aspects are applicable to a range of applications, and may be applied to multiple (individual or sets of) classes, otherwise not necessarily related within an application. This has to do with the AspectJ view of the world that aspects cross-cut *the* class structure. That is, they are organized around a common class structure. In fact, AspectJ aspects lack their own class structure. They are, rather, an assembly of weave instructions. These different module structures underlying applications and aspects is reflected also in the AOP terminology, where there are components and there are aspects. A component can be encapsulated in a generalized routine, while an aspect cannot. It follows that an aspect is not and cannot be a component.

In this paper, we challenge this conclusion. We observe that components, specifically, a variation of *Adaptive Plug&Play* (AP&P) components [22], are useful not only in modeling behavioral composition but, as we shall demonstrate, they are also good constructs for expressing aspects. We call this new kind of component an *aspectual component*. According to our view of the world, aspects have their own class structures which mutually cross-cut each other. We put the emphasis on genericity. A construct for aspect definition ought to enable us to write slices of functionality without committing ourselves to other depending slices or to the base object structure with which it is going to be deployed.

The aspectual component model, which we propose in this paper, is a module for capturing aspects that complements classes in precisely this way. Aspects in an aspectual component are written to their own class graph referring to abstract *join points* when needed which constitute the expected interface of aspectual components. Resolving join points is done apart of aspect definitions in explicit *connector* constructs. Connectors map the class graph of one aspect to the class graph of the other. Mapping constructs make use of *declarative* techniques such as pattern matching on operation names and *Adaptive Programming* (AP) in order not to commit to irrelevant details of the target graph. We show that not only are aspects expressed separately and in a modular way via aspectual components: they can be easily reused in new contexts due to the separate expression of the weaving process. In the absence of a precise definition for AOP, let alone a formal one, we follow the AOP tradition started by Kiczales and colleagues in their seminal paper on AOP [11]: we present our ideas in an example-driven manner; and, to make an appealing case, we demonstrate how we can re-implement the examples found in the AspectJ tutorial using our aspectual components. Furthermore, we present an implementation strategy that allows weaving of aspectual components that are available only in binary form, using binary component adaptation [**?**].

3 The Aspectual Component Model

In this section, we propose a component model for *Aspect-Oriented Programming* (AOP). We illustrate the basic concepts of this model with a small example. We then introduce the full model by presenting new concepts using increasingly more sophisticated examples. A substantial part of the examples are based, slightly modified, on the *AspectJ* tutorial [27]. AspectJ is an implemented language extension created by the AOP mentors. It targets at bridging the gap between object and aspectual decomposition in Java.

3.1 Aspectual Components in a Nutshell

The functionality captured by an *aspectual component* is written in terms of its own formal class graph, called *participant graph* (PG). This minimizes the knowledge about the structure and functionality of the application as well as other aspects with which it is going to be combined.

An aspectual component consists of a set of *participants* serving as the nodes of the PG. A participant is a class in the PG, which stands at the same level as a formal argument in a procedure declaration. It needs later to be bound to classes in other PGs or to a concrete *class graph* (CG). The PG defines structural constraints (the equivalent to type constraints in formal arguments) that must be satisfied by the *actual* participating classes. The constraints to be satisfied are defined later. A participant lists a set of operation signatures it expects other aspects to supply, preceded by the keyword **expect**. Expected operations are used/modified in the aspectual component-specific definition of the participant.

Definition 3.1 (Aspectual Component) *An aspectual component is*

1. *a set of participants forming a graph called the PG (represented, e.g., by a UML class diagram.) A Participant is a formal argument which consists of:*
   - *expected features (keyword **expect**)*
   - *reimplementations (keyword **replace**)*
   - *local features (data and operations.)*
2. *aspectual component-level definitions*
   - *local classes, visible only within the aspectual component*
   - *features (data and operations; there is a single copy of each global data member for each deployment)*

For illustration, consider how our read access monitoring example is captured by the aspectual component ShowReadAccess below. The code in ShowReadAccess is actually as free of details about concrete deployment contexts as was the informal English description. It simply states that whatever actual class(es) in an application will be bound to the single participant in the aspect definition, DataToAccess, and whatever method(s) in the actual class(es) will be assigned to implement the interface expected from DataToAccess, readOp in this case, the latter methods will be replaced (the replace statement below) by an implementation that first prints out a message and then executes the original method (the expected() invocation below).

```
package ShowAccess;
component ShowReadAccess {
  participant DataToAccess {
    expect Object readOp();
    replace Object readOp() {
      System.out.println("Read access on " + this.toString());
      return expected(); // the expected readOp()
      }
    }
  }
```

Here ShowAccess is a name space in which only the ShowReadAccess component is defined. This component can refine a single method of some class. The keyword **participant** denotes that DataToAccess is a formal class name, which will be bound to an actual class when the component is applied. The formal participant lists the methods it expects the actual participant class to have. When the component is connected, some actual method(s) of an actual class would be mapped to DataToAccess.readOp.

Except from what it expects, the component also declares what it provides. In this example, it provides a different implementation for whatever method is going to bind readOp. This is expressed by the keyword **replace**. The call *expected()* invokes the old implementation, and resembles the use of *super()* when overriding a method in class inheritance, and *this()* when calling another constructor in Java.

The meaning of the pseudo-variable this is the same as in a Java program. In the definition above, it denotes the application's instance being instrumented. If there are no ambiguities, there is no need to explicitly refer to the pseudo-variable this. Any invocation without an explicit receiver is considered to be a self-invocation. A method for this invocation is first looked-up in the aspect-specific definition of the participant. If no method is found there, the class in the application assigned to play the participant role is searched for.

Since each aspectual component has its own PG, when composing two components together and/or deploying them with a concrete object structure, we need to map PGs one to the other and/or project them to a concrete CG. That is, we need to resolve the interdependencies between different slices to construct the whole. This is done apart of the definition of an aspect by means of *connector* constructs. Broadly speaking, a connector consists of several statements of the form: "ClassSet is ParticipantSet with ExpectedMappings, specifying which class(es) in one PG/CG bind which participant(s) in the other PG and corresponding methods that implement the expected interface(s). For illustration consider a trivial application that models graphical shapes (the Shapes package below) and consider a deployment scenario for ShowReadAccess in which we are interested in instrumenting only read accesses on point objects, specified by the ShowReadAccessConn1 connector (the Deployment package below.)

```
package Shapes;
class Point {
  private int x = 0;
  private int y = 0;
  void set(int x, int y) { setX(x); setY(y); }
  void setX(int x) { this.x = x; }
  void setY(int y) { this.y = y; }
  int getX(){ return this.x; }
  int getY(){ return this.y; }
}
class Line { ... }
class Rectangle { ... }

package Deployment;
import ShowAccess.*;
import Shapes.*;
connector  ShowReadAccessConn1 {
   Point is ShowReadAccess.DataToAccess with {readOp = get*};
}
```

The connector in the example above is simple. It contains only one class–to–participant mapping. As already mentioned, in general, there may be several such mappings in a connector. In addition, the class–to–participant mapping is in general many–to–many. That is, several classes might be bound to the same participant, or the other way around, a single class may be bound to several participants, i.e., the class may appear on the left hand side of several is–statements. Furthermore, a class can be bound to a participant with multiple sets of ExpectedMappings. In the example above the mapping is done based on pattern matching on operation names, with the pattern being specified by the regular expression `get*`. In general, this mapping can also take a more "programmatic" form, i.e., an implementation of the expected signature is given by means of composing functionality provided by the application. Later in this section, we will present more sophisticated examples that give a more complete picture of the structure of connectors. As it will be illustrated later, connectors can also be used to map the provided interface of some aspectual components to the expected interface of some others, in order to build composed aspects.

3.2 The Key Idea

At this point, let us summarize the key idea behind aspectual components as a construct for reconciling object-based and aspect-based decomposition. Aspectual components add a new dimension in the organization of object-oriented software, as illustrated in Fig. 2. First, software is decomposed based on aspects which are symbolically represented by rectangles with rounded corners. Different grey scales are used in Fig. 2 to distinguish different aspects. Second, each aspect definition has its own object-based decomposition. Each object type involved in an aspect definition is represented by a shape in Fig. 2. The sets of shapes used in different aspects are disjoint to illustrate that each aspect is written modulo to its own class graph.

The expected interface of aspect definitions is represented by the transparent (colorless) shapes on the left hand side of aspect–areas in Fig. 2. Shape towers on the right hand side of each aspect represent the provided interface of the aspect. The modifications and/or extensions of the expected interface by aspect definitions are illustrated by replacing colorless shapes on the left hand side by colored shapes on the right hand side. In addition, some new shapes are added to each tower on the right hand side as compared to the corresponding tower on the left hand side. The left most (white) area in Fig. 2 represents a concrete application. It is self-contained in that it does not expect anything to be supplied from the outside and provides the basic implementation of the application functionality. It is generally much richer in data type definitions than the aspect definitions.

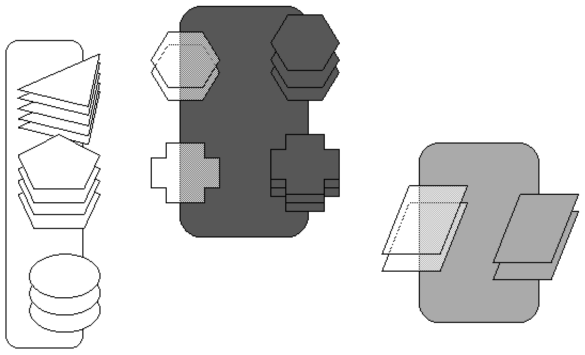

Figure 2: Aspect Decomposition with Components

The deployment process is schematically presented in Fig. 3. On the left hand side of the figure, the definition of an application with three classes and the definition of an aspect with one participant is schematically presented. The connector is symbolically represented by the two colored rectangle between the application and the aspect in the deployment part of the figure. It binds two application classes to the single participant of the aspect. The result of compiling together application, aspect and connector definitions is equivalent to the set of shape towers on the right hand side of Fig. 3. Partial definitions from the application and the aspect are mixed together into the final definition of the objects represented by pentagon and circles (the classes that are bound to the aspect's single participant).

Composing aspectual components with each other is similar to deploying a component with an application. The result is in this case a new composed component with its expected and provided interfaces resulting from the connection. As already mentioned, in the aspectual component model, there is in fact no strict separation between aspects (aspectual components) and applications they get deployed with. An application is just a special kind of an aspectual component that do not have an expected interface. The difference

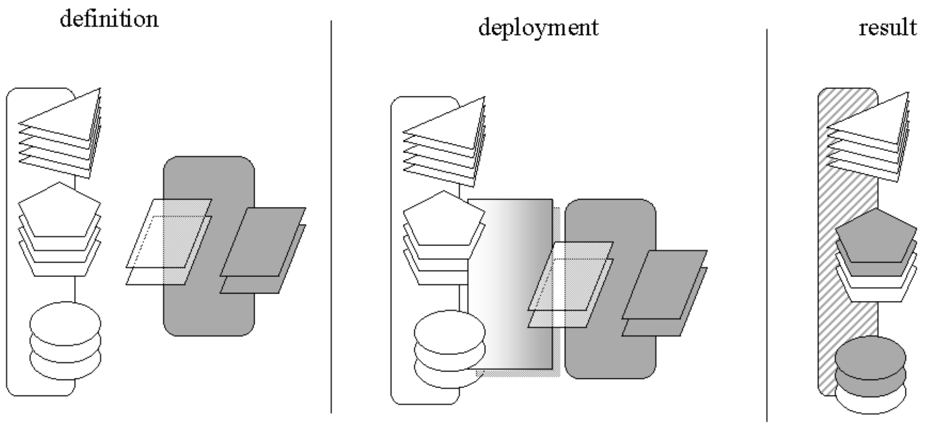

Figure 3: Deploying Aspects with Applications

between deploying an aspect with an application as opposed to composing two aspects is not in the process how this happens. Rather in what is the result. The deployment produces weaved running code, while composition produces new components with still abstract participants. These issues will be illustrated in the following section.

Specifying deployment details apart of an aspect definition, make the aspect reusable with other classes within the same application, or other applications without changing its definition. What is needed is the definition of new connectors. For illustration, assume that after having defined ShowReadAccessConn1, we realize that read accesses on lines need to be instrumented as well. For this purpose the ShowReadAccessConn2 connector is defined below as an extension of ShowReadAccessConn1.

```
connector ShowReadAccessConn2 extends ShowReadAccessConn1  {
  Line is DataToAccess with {readOp = get*};
}
```

On the other side, if we knew from the very beginning that whatever classes are included in the application, we would like to instrument all read access operations, we would have defined ShowReadAccessConn3, rather than ShowReadAccessConn1. If this was the case, no modification of the aspect or extension of the connectors would be needed, if we extend the application with a new data type, say Circle.

```
connector ShowReadAccessConn3 {
  Shapes.* is ShowReadAccess.DataToAccess with {
    readOp = get*;
  }
}
```

An aspect can be defined to affects several operation categories of the same class in different ways. For illustration, assume that in addition to read accesses, we would like to instrument write accesses. This is expressed by the ShowReadWriteAccess component below. The aspect is then deployed with the shape package by ShowReadWriteAccessConn1.

```
package ShowAccess;
component ShowReadWriteAccess {
  participant DataToAccess {
     expect Object readOp();
     expect void writeOp(Object[] args);
     replace Object readOp() {
        System.out.println("Read " + displayString());
        return expected();
```

```
      }
     replace void writeOp(Object[] args) {
        System.out.println("Write " + displayString());
        expected(args);
      }
     private String displayString() {
       return "access on " + this.toString();
     }
   }
 }

package Deployment;
import ShowAccess.*;
import Shapes.*;
connector ShowReadWriteAccessConn1 {
   Shapes.* is DataToAccess with {
      readOp = set*;
      writeOp = get*;
  }
}
```

Assuming that ShowReadAccess was already defined, we could also define ShowReadWriteAccess as an extension of ShowReadAccess, and the corresponding connector as an extension of the ShowReadAccessConn3 connector, thus supporting reuse at the level of both aspect and connector definitions.

```
component ShowReadWriteAccess extends ShowReadAccess {

  participant DataToAccess {
     expect void writeOp(Object[] args);

     replace void writeOp(Object[] args) {
        System.out.println("Write access on " + this.toString());
        expected(args);
     }

   }
 }

connector ShowReadWriteAccessConn2 extends ShowReadAccessConn3  {
  Shapes.* is DataToAccess with {
    writeOp = set*;
}
```

3.3 Aspectual Components versus Inheritance

Before going into further details about using aspectual components for aspect-oriented programming, let us first discuss how the behavior composition mechanisms underlying aspectual components differs from that underlying inheritance. The definition of ShowReadWriteAccess in the previous subsection suggests a strong similarity to the definition of an abstract class. So one might ask, why not define the ShowReadWriteAccess aspect as an abstract class, as shown below, and use inheritance to deploy the aspect with the application. The application would subclass the abstract class delegating the expected operations. Aspects with multiple participants would then correspond to "small" frameworks.

```
abstract class ShowReadAccess {
  abstract Object expectedReadOp();
  public Object providedReadOp() {
    System.out.println("Read access on " + this.toString());
    return expectedReadOp();
  }
}
```

The main difference between the two mechanisms is that with aspectual components the aspects can be written independently from each other and from the applications, while with the framework approach the application has to be aware of the aspects before hand. Furthermore, with aspectual component the code that connects building blocks together is not part of any individual building block, while the opposite is true for the framework approach. The connection (adaptation) code would be part of the application. The independence of building blocks from each other allows independent development and compilation. On the other hand the separate connection code allows reusing the same application with different aspects and vice versa, or many–to–many mappings of actual classes/operations to participants. For instance, there is no way we can map all set* operations of the class Point to the expected readOp of the aspect by making Point a subclass of ShowReadAccess.

If modeled as an abstract class, the applicability of ShowReadAccess will be restricted to a single operation in each application class to be instrumented. Not to mention that modeling the instrumentation aspect as a class and making Point (or shapes in general) a subclass of ShowReadAccess is not conceptually clean. Conceptually classes correspond to abstractions in the application domain, which is not true for ShowAccess aspects. On the other hand there is no is–a relationship between Point and ShowReadAccess. We will end up establishing spurious subclassing relationships between application classes and aspect classes. The whole picture gets much more complicated if we think of composing aspects together which also depend on each other.

4 Composing Aspects

In the examples so far, we've illustrated only the basic elements of the *aspectual component* model, namely, aspect definition, deployment, and reuse. In the example to follow, we show more details of aspect definition, concentrating on the manner aspects are composed. The first example presents two aspects that are composed by being deployed simultaneously within the same application.

The first aspect, called InstanceLogging, describes the logging of new instances in a system. The InstanceLogging component demonstrates the details of definition, initialization, and use of a shared aspect-related feature. Application classes, that will be assigned to participant roles of the aspect in a certain deployment, will all share the aspect-state. For instance, regardless of how many application classes are be assigned to DataToLog during a deployment, they will all share the logObject variable defined in InstanceLogging.

```
component InstanceLogging {
  participant DataToLog {
    expect public DataToLog(Object[] args);
    replace public DataToLog(Object[] args) {
      expected(args);
      long time = System.currentTimeMillis();
      try {
        String class =  this.class.getName() + " ";
        logObject.writeBytes(""New instance of " + class + at "" " + time + "" " \n");
      } catch (IOException e) {System.out.println(e.toString());}
    }
  }
  private DataOutputStream logObject = null;
  static {
    try {
      logObject = new DataOutputStream(new FileOutputStream(log));}
      catch (IOException e) {System.out.println(e.toString());}
    }
}
```

The second aspect describes the automatic resetting of certain data types values after a reaching a certain number of accesses count. This second aspect, called AutoReset, is defined as follows.

```
component AutoReset {
  participant DataToReset {
    expect void setOp(Object[] args);
    expect void reset();
    protected int count = 0;
    replace void setOp(Object[] args) {
     if ( ++count >= 100 ) {
        expected(args);
        count = 0;
        expected();
       }
     }
   }
}
```

This aspect introduces a new instance variable, called count, in each application object that will be mapped to the DataToReset participant role in the component. It is an instance variable, since it is declared as non-static within the aspect-specific definition of DataToReset. The AutoReset component illustrates another feature of aspectual components: there are two kinds of expected operations. The first kind, represented by setOp, includes operations which are expected in order to be replaced by the aspect (expected operations preceded by the keyword replaced). Operations of the second kind, represented by reset, are operations expected in order to be used by the aspect-specific definition of the participants. Expected operations of the first kind constitute the *modification interface* between aspects and application. This corresponds to the concept of joint points in *AspectJ*. Expected operations of the second case constitute what we call *usage interface* between aspects and application.

Let us now consider how these aspects are composed by being deployed with the Shapes application, we have been using so far. Assume that we want (a) all three aspects to apply to points, (b) ShowReadWriteAccess and InstanceLogging to apply to Line, and (c) only InstanceLogging to apply to Rectangle. This is expressed by the connector CompositionConn1 below. This connector illustrates that besides specifying mappings for expected operations by means of pattern matchings on method names, the deployment programmer can also directly implement an operation in the expected interface in terms of operations and/or state defined in the application. This is illustrated by the definition of reset() in CompositionConn1, i.e., in this case the mapping is "programmatic" rather than declarative.

```
connector CompositionConn1 {
    {Line, Point} is ShowReadWriteAccess.DataToAccess with {
          readOp = get*;
          writeOp = set*;
       };
   Point is AutoReset.DataToReset with {
         setOp = set*;
         void reset() { set(0, 0);}
      };
   Shapes.* is InstanceLogging.DataToLog;
}
```

The composition structure expressed by CompositionConn1 is symbolically represented in Fig. 4 and 5. The composed deployment is presented on the left hand side of Fig. 5. On the right hand side the result of the deployment is shown by the towers of shapes, representing the resulting weaved code for points (represented by triangles in the figure), lines (represented by triangles in the figure), respectively rectangles (represented by circles in the figure).

Specifying connections apart of the definitions of the individual aspects, makes aspect definitions more reusable. For instance, we can later add new connection statements to specify additional "joint points"

connector CompositionConn1			
Shapes	.Point	.Line	.Rectangle
ShowReadWriteAccess.DataToAccess	x	x	
AutoReset.DataToReset	x		
InstanceLogging.DataToLog	x	x	x

Figure 4: Connector graph for CompositionConn1

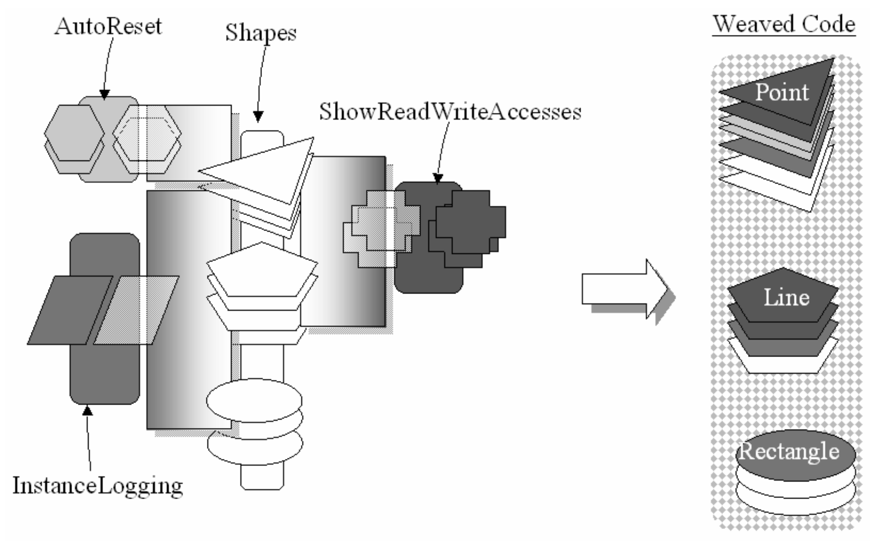

Figure 5: Deploying Several Aspects with the Same Application

between an application and aspects, as illustrated by the CompositionConn2 connector below, which deploys the AutoReset aspect also with lines.

```
connector CompositionConn2 extends CompositionConn1 {
  Line is AutoReset.DataToReset with {
    resetOp = set*;
    void reset() {init();}
  };
}
```

Besides adding new connection statements, one can also modify existing ones. This is illustrated by the connector CompositionConn3 below. It redefines the connection CompositionConn1's statement for Point, to distinguish between the reset policy for operations that change both coordinates simultaneously and operations that change only one of the coordinates. Three different cases are distinguished by specifying three different *connection clauses*. This connection statement overrides the respective connection statement for the class Point, in CompositionConn1.

```
connector CompositionConn3 extends CompositionConn1 {

   Point is AutoReset.DataToReset with {
      { setOp = set;
        void reset() { set(0, 0);}
      }

      { setOp = setX;
        void reset() { setX(0);}
      }

      {
        setOp = setY;
        void reset() { setY(0);}
      }
   };

 }
```

In the example considered so far, the aspects to be composed are independent from each other. The connector CompositionConn1 is equivalent to specifying three independent connectors (actually this is what Fig. 5 suggests). The order of deploying the individual aspects being composed is not important. In the following, we will illustrate how the expected interface of an aspect can be connected to the provided interface of another aspect after the latter has been itself deployed with classes in a concrete application. Consider first the following two components.

```
package aspect;
component DataWithCounter {
   protected class Counter {
       int i = 0;
       void reset() {i=0;}
       void inc() {i++;}
       void dec() {i--;}
   }
   participant DataStructure {
       protected Counter counter = new Counter();
       expect void make_empty();
       expect void push(Object a);
       expect void pop();
       replace void make_empty() {
           counter.reset();
```

```
            expected();
        }
        replace void push(Object a) {
            counter.inc();
            expected(a);
        }
        replace void pop() {
            counter.dec();
            expected();
        }
     }
}
component DataWithLock {
  participant Data {
    Lock lock = new Lock();
    expect AnyType method_to_wrap(Object[] args);
    replace AnyType method_to_wrap(Object[] args) {
      if (lock.is_unlocked()) {
        lock.lock();
        expected(args);
        lock.unlock(); }
      }
    }
  protected class Lock {
    boolean l = true;
    void lock() {l = false;}
    void unlock() {l = true;}
    boolean is_unlocked() { return l;}
  }
}
```

The first aspect adds functionality for counting the elements included in a container data type, such as stacks, queues, etc. The second aspect adds functionality for synchronizing invocations of certain operations on a given data type by means of a lock. Both components nest the definitions for counter, respectively, lock, which are private to them. Now assume an application which provides implementations for data types, such as stacks, queues, etc. We can now model stack and queue objects that count their elements and the operations of which are synchronized by means of a lock by specifying the connector addCounter&Lock in the deployment package below. This kind of composition is schematically presented in Fig. 6[1].

```
package appl;
class StackImpl {
   String s = new String();
   void empty() {s = "";}
   void push(char a) {s = String.valueOf(a).concat(s);}
   void pop() {s = s.substring(1); }
   char top() { return (s.charAt(0));}
 }
class QueueImpl {...}

package deployment;
connector addCounter&Lock {
  StackImpl is DataWithCounter.DataStructure with {make_empty = empty;}
                                          //the rest of the mapping is implicit
                                          // since the rest of operations in the
                                          // modification interface of the aspect
                                          // have the same name as the application
```

[1]Deploying the aspects with QueueImpl is omitted in the figure as compared to the addCounter&lock connector for the sake of simplicity

```
                                    // operations to be bound to them
                 is DataWithLock.Data with {
                    method_to_wrap = {pop, push, top, make_empty}
                  };
  QueueImpl implements DataWithCounter.DataStructure with {
     ... } is DataWithLock.Data with { ... };
}
```

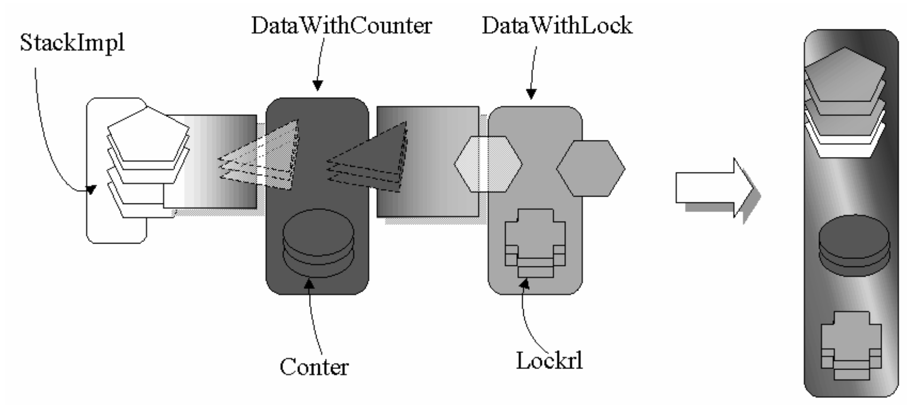

Figure 6: Deploying Several Aspects with the Same Application (2)

Alternatively, one can create composed aspects prior to deployment with any concrete application by connecting part of the expected interface of one aspect to part of the provided interface of another. The remaining unbound parts of the expected interfaces are connected to applications during deployment. Creating composed aspects prior to deployment is illustrated by the following code. First, DataWithStack is composed with DataWithLock into a new aspect, DataWithCounter&Lock. Later DataWithCounter&Lock get deployed with the concrete application. The process of creating DataWithCounter&Lock is schematically presented in Fig. 7.

```
package aspect;
component DataWithCounter&Lock {
   participant Data = DataWithCounter.DataStructure is DataWithLock.Data with {
        method-to-wrap =  {make_empty, pop, top, push}};
}

package deployment;
connector addCounter&Lock {
  StackImpl is DataWithCounter&Lock.Data with {
       void make_empty {empty;}
       }
  };
  QueueImpl is DataWithCounter&Lock.Data with { ...};
}
```

4.1 Defining New Behavior: The Publisher-Subscriber Aspect

In the examples considered so far, there has always been a single deployment of an aspect to a certain application. In the following we will illustrate that due to separate deployment specifications an aspect can be multiply deployed with the same application, each deployment with its own mappings. For illustration, a publisher/subscriber protocol aspect is modeled in the component below.

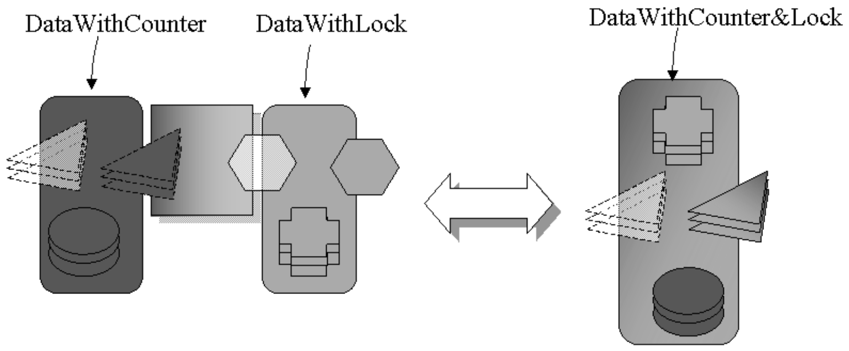

Figure 7: Deploying Several Aspects with the Same Application (3)

```
component PublisherSubscriberProtocol {

   participant Publisher {
       expect  void changeOp(Object[] args);
       protected Vector subscribers = new Vector();
       public void attach(Subscriber subsc) {
           subscribers.addElement(subsc);}
       public void detach(Subscriber subsc) {
           subscribers.removeElement(subsc);}
       replace void changeOp() {
          expected();
          for (int i = 0; i < subscribers.size(); i++) {
            ((Subscriber)subscribers.elementAt(i)).update(this);}
        }
   participant Subscriber {
      expect void subUpdate(Publisher publ);
      protected Publisher publ;
      public void newUpdate(Publisher aPubl) {
         publ = aPubl;
         expected.subUpdate(publ);}
      }
   }
}
```

In addition to replacing existing methods with enhancements, PublisherSubscriberProtocol defines new fields and methods that are involved in the protocol and establishes the structural relationship between the protocol participants. Note that the protocol in PublisherSubscriberProtocol is defined for two hypothetical participants and not in terms of particular classes. As such, it can be applied to different publisher/subscribers pairs, as illustrated by the following two connectors. In the first case, the publisher role is played by the class Point, while the subscriber role is played by a simple class, called ChangePrinter, shown below. Establishing the publisher/subscriber protocol between Point and ChangePrinter is realized by the connector PubSubConn1.

```
package Output;
class ChangePrinter {
  void public printR() {
     System.out.println("Printer: " + this.toString() +
                  " read access has occurred ..." + \n);
   }
  void public printW() {
     System.out.println("Printer: " + this.toString() +
                   " write access has occurred ..." + \n);
   }
  void public notifyChange() { System.out.println("CHANGE ...");
  }
}

package Deployment;
connector PubSubConn1 {
  Point is Publisher with
     { changeOp = {set*, get*};}
  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
         notifyChange();
         System.out.println("on point object " + ((Point) publ).toString());
       }
   }
 }
```

In a similar way, the following code, illustrates the deployment of the protocol defined by the PublisherSubscriberProtocol aspect with another application: this time a game package including a class for the TicTacToe game and a class for the graphical interface, GUI, with the latter being an aggregate of two other classes, BoardDisplay and StatusDisplay, as shown below.

```
connector PubSubConn2 {
  TicTacToe is Publisher with {
    changeOp = {startGame, newPlayer, putMark, endGame}};

  {BoardDisplay, StatusDisplay} is Subscriber with {
       void update(Publisher publ) {
          setGame((Game) publ);
          repaint();
        }
     };
 }
```

Both deployments of the publisher/subscriber aspect shown so far, result in the standard publisher/subscriber protocol, i.e., the protocol with one publisher being subscribed by a single set of subscribers, uniformly interested in any change in the publisher's state. One can also realize modified versions of the standard protocol by simply defining new connectors. For instance, the connectors, PubSubConn3 and PubSubConn4, deploy the protocol with the same classes as PubSubConn1, but now two different connection clauses distinguish the notification operations that are invoked on the subscribers depending on which access operations are invoked on the publisher. This corresponds to a variation on the publisher/subscriber pattern, where there are several lists of subscribers for one publisher, distinguished by the events they are interested in.

```
connector PubSubConn3 {

  Point is Publisher with { changeOp = set*;}

  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
          printW();
          System.out.println("on point object " + ((Point) publ).toString());
          }
       }
}

connector PubSubConn4 {
  Point is Publisher with { changeOp = get*;}

  ChangePrinter is Subscriber with {
        void subUpdate(Publisher publ) {
          printR();
          System.out.println("on point object " + ((Point) publ).toString());
        }
     }
 }
```

The sets of operations of Point that are mapped to different notification operations of the subscriber participant need not be disjoint. For instance, we may want to distinguish between set operations that affect the x-coordinate, respectively, the y-coordinate of a point. The set(int, int), however, will then fall in both categories. This is expressed by the connectors PubSubConn3.1 and PubSubConn3.2 below.

```
connector PubSubConn3.1 {

  Point is Publisher with { changeOp = {set, setX};}

  ChangePrinter is Subscriber with {
       void subUpdate(Publisher publ) {
          printW();
          System.out.println("on the X coordinate of " +
                + "point object " + ((Point) publ).toString());
          }
     }
}

connector PubSubConn3.2 {
  Point is Publisher with { changeOp = {set, setY}; }
  ChangePrinter is Subscriber with {
      void subUpdate(Publisher publ) {
          printW();
          System.out.println("on the Y coordinate of
                point object " + ((Point) publ).toString());
          }
   }
}
```

4.2 Mapping Participant Graphs

Generalizing over our experience with programming using class graphs, which are named, in earlier work, *interface class graphs* or *strategy graphs*, we extracted the property *refinement-instance*. This property must hold between a *participant graph* (PG) and a corresponding *class graph* (CG) or another PG. The intent of the refinement-instance relation is to ensure that the behavior in the component will be properly instantiated at the place of use without “surprising” behavior. The refinement-instance relation can be checked efficiently using algorithms from Adaptive Programming [23, 15]. The refinement-instance check will be a part of the aspectual component compiler using the AP library described in [15]. In the following, we assume that the PGs have only one kind of nodes and one kind of edges. Generalization to PGs that have the features of UML class diagrams, such as abstract and concrete classes and directed associations and inheritance is described in [23, 15]. For connection to be possible, a PG and the target graph to which the PG is mapped, must be in an instance relationship. Informally, a graph G is an instance of graph S if G contains the connectivity of S. We think of S as an abstract graph and G as a concrete graph. Formally, a graph G is an instance of a graph S, if S is a connected subgraph of the transitive closure of G.

Informally, a graph G is a refinement-instance of a graph S if the connectivity of S is in a "pure form" in G. By *pure form* we mean that there are no "surprising uses" of the nodes of S. Formally, a graph G is a refinement-instance of a graph S, if S is a connected subgraph of the pure transitive closure of G with respect to the node set of S. The pure transitive closure of $G = (V, E)$ with respect to a subset W of V is the graph $G* = (V, E*)$, where $E* = (i, j)$: there is a W-pure path from vertex i to vertex j in G. A W-pure path from i to j is a path where i and j are in W and none of the inner points of the path are in W.

5 Implementation Issues

A straightforward way to implement the aspectual component model is to translate aspectual components into AspectJ aspects, expand all the connectors and tangle the aspects with connector information. However, the problem with using AspectJ as the underlying implementation technology is that it is based on source code modification, thus implying that the source code is available for all three: application classes, aspectual components, and connectors. We find, however, that it is highly desirable to allow separate compilation of application and aspectual components, i.e., a true separation of definition and deployment processes. For this reason, our attempt has been focused on implementation strategies that cope with the requirement that application and aspectual components are pre-compiled units available only in binary form. In the following, we outline the essentials behind an implementation strategy that fulfills this requirement.

The implementation strategy we outline here assumes that aspectual components are written in standard Java syntax. In Sec 4 we discussed the composition model underlying aspectual components versus inheritance. In conjunction with this discussion we indicated that the participants in an aspectual component can actually be written as abstract classes in pure Java, whereby expected operations in both the component's usage and modification interface are modeled as abstract methods. For the implementation technique outlined here, we assume a slightly modified implementation of aspectual participants as abstract classes. In this implementation, the set of abstract methods includes only expected operations in the usage interface of the aspectual participant, but not operations in the modification interface.

Remember that the modification interface consists of those expected operations that are replaced by the aspect, i.e., there is an aspect definition of the same operation preceded by the keyword replace. This keyword is omitted in the pure Java implementation of an aspectual component by being inlined in the name of modification operations. If op is an expected operation that is replaced in an aspect, in the aspectual component language for aspects, we would have replace Type op(argTypes) { ... }. In the pure Java version of the same aspectual component, we would have an operation, named replaced_op. Instead of being declared as an abstract method, the expected implementation of op, will be modeled as a parameter to replaced_op.

For illustration, consider the implementation of the aspectual component AutoReset from Sec. 4 using this style in the class AutoReset below. Remember, reset was in the usage interface of AutoReset. Hence, it is modeled as an abstract method in the class AutoReset.DataToReset. This is different with the expected operation setOp which is replaced by the AutoReset aspectual component in Sec. 4. As already discussed above, there will be an operation, replaced_setOp that implements how the aspect modifies the expected setOp. The formal parameter, expected_setOp of replaced_setOp will be bound to implementations expected from applications. In the implementation of replaced_setOp the expected-invocation (expected()) is replaced by sending an invoke message to expected_setOp passing thisObject and args as the receiver, respectively the arguments of the invocation.

```
class AutoReset {

  abstract class DataToReset {
    abstract void reset();

    protected int count = 0;

    void replaced_setOp(Object thisObject, Class thisClass,
                  Method expected_setOp, Object[] args) {
     if ( ++count >= 100 ) {
        expected_setOp.invoke(thisObject, args);
        count = 0;
        reset();
       }
     }
   }
}
```

In addition to the original implementation expected from the application (the method object expected_setOp), other parameters of replaced_setOp include: (a) the receiver of the original application operation (thisObject), (b) the class of this object (thisClass), and the arguments to the original message call (args). This is a set of standard parameters passed to any operation that implements an aspect modification of some expected functionality. This set provides not only the information needed for invoking the original implementation, but also supplies some meta-information about the application which might be used within the aspect definition.

So far, we showed that aspect components themselves can be written in pure Java. This is not surprising given that what is really new about the aspectual component model are the connectors which realize a composition model that cannot be mapped to standard Java in a straightforward way. It is here that extra tool support is needed. By tool support we mean, that connector specifications need to be processed by a pre-compiler. Given the application/aspect binaries as well as the connector specifications, the pre-compiler conducts the following two main steps.

Binary Adaptation of Application Classes

First, binaries of those application classes that are mapped to an aspectual participant, are adapted by applying binary component adaptation techniques, presented in [?]. The binary adaptation consists in the following.

- Application classes are turned into event publishers by adding a field that stores a set of subscribers. Subscribers will be instances of classes generated from the translation of connector specifications (see step two).
- Any application operation, op, which the connector maps to an expected operation in the modification interface of the aspectual component is renamed to expected_op.
- A new implementation for the original signature, op, is added. The new method simply invokes a notify message on the subscribers passing along the receiver of the message, the definition class, the method object for expected_op (i.e., the original implementation), and an array containing the arguments to op.

Pseudo-code is given in the class Point below to illustrate how the binary of the class Point in the Shapes application will be adapted as the result of compiling the CompositionConn3 connector given in Sec. 4.

```
class Point {
    public static java.util.Vector aspectSubscribers; // added variable

    public void addSubscriber(AspectSubscriber sub) {  // added operation
        aspectSubscribers.addElement((Object) sub);}

    private int x = 0;
    private int y = 0;

    void expected_set(int x, int y) {       // renamed
      this.x = x; this.y = y;}

    void set(int x, int y) {           // reimplemented
      Object[] args = {(Object) new Integer(x), (Object) new Integer(y)};
      Class[] argTypes = {Integer.TYPE, Integer.TYPE};
      Method expected_set = thisClass.getMethod("expected_set", argTypes);
      Enumeration subscribers = aspectSubscribers.elements();
      while (subscribers.hasElements()) {
         Object sub = subscribers.next();
         sub.notify(this, this.getClass(), expected_set, args);
       }
     }


    void expected_setX(int x) { this.x = x; } // renamed
    void setX(int x) { ... }  // reimplemented similar to set

    void expected_setY(int y) { this.y = y; } // renamed
    void setY(int y) { ... } // reimplemented similar to set

    int getX(){ return this.x; }
    int getY(){ return this.y; }
  }
```

Generating Connector Classes

The second step conducted by the pre-compiler is to generate a class based on connector specifications. For illustration, the class generated for CompositionConn3 in Sec. 4 is given below. Remember that CompositionConn3 contains an is-statement, mapping the class Shapes.Point to the participant AutoReset.DataToReset with three different ExpectedMappings clauses. These clauses map the participant's modification interface (consisting of a single operation, setOp) to three different application methods (set, setX, and setY respectively), specifying each time a different implementation for the usage interface (the reset operation).

Connector objects use a hashtable in order to associate incoming notify events fired by the adapted application with invocations of respective aspect modification operations. In the CompositionConn3 below this is the instance variable mappings. As indicated also by the initialization of this table in the implementation of the constructor CompositionConn3(), the keys of a connector's mappings table are the application methods that are modified by the aspect participant(s). For any Class is Participant statement and operation replaced_op in the modification interface of Participant, there will be an object for each ExpectedMappings clause of the is-statement. These objects are defined as anonymous inner objects of a subtype of Participant within the connector. These inner objects will be the values in the mapping hashtable. They share two definitions which are defined in a common superclass.

- an instance variable for storing the application object receiving a message modified by the aspect, and
- an operation, called eval, that expects the same parameters that are passed along a notify event by the application. Eval sets the host instance variable and invokes the replacement method, replaced_op on itself (remember, this is a subclass of Participant.

Given the Class is Participant statement and operation replaced_op in the modification interface of Participant, different ExpectedMappings clauses differ from each other in that they associate different implementations of the usage interface with different mappings of replaced_op to application methods. As a result, the inner objects corresponding to different ExpectedMappings clauses will extend the common superclass by providing different implementations for the usage interface.

```
interface AspectSubscriber {
   void notify(Object thisObject, Class thisClass, Method expected_meth, Object[] args);
 }

interface Evaluable {
   void eval(Object thisObject, Class thisClass, Method expected_meth, Object[] args);
 }
```

```
// connector class generated from the connector specifications

class CompositionConn3 implements AspectSubscriber {

   public static singleInstance = new CompositionConn3();
   java.util.Hashtable mappings = new java.util.Hashtable();

   abstract class Point_DataToReset_setOp extends AutoReset.DataToReset implements Evaluable {

     private Point host;

     public void eval(Object thisObject, Class thisClass, Method expected_meth,
             Object[] args) {
        host = (Point) thisObject;
        replaced_setOp(thisObject, thisClass, expected_meth, args);
      }
    }

   public CompositionConn3() {
     Object[] argTypes = {Integer.TYPE, Integer.TYPE};
     Method expected_set = Point.getMethod("expected_set", argTypes);
     argTypes = {Integer.TYPE};
     Method expected_setX = Point.getMethod("expected_setX", argTypes);
     Method expected_setY = Point.getMethod("expected_setY", argTypes);

     mappings.put((Object) expected_set,
                  (Object) new Point_DataToReset_setOp() {
                             void reset() {host.expected_set(0, 0);}
                           });
     mappings.put((Object) expected_setX,
                  (Object) new Point_DataToReset_setOp() {
                             void reset() {host.expected_setX(0);}
                           });
     mappings.put((Object) expected_setY, (Object)
                  (Object) new Point_DataToReset_setOp() {
                             void reset() {host.expected_setY(0);}
                            });
     Point.addSubscriber(this);
     }

   public void notify(Object thisObject, Class thisClass,
                      Method expected_meth, Object[] args) {
      if (mappings.containsKey((Object) expected_meth) {
        Evaluable participant = (Evaluable) mappings.get((Object) expected_meth);
        participant.eval(thisObject, thisClass, expected_meth, args);
          }
      }

}
```

For instance, the "Point is DataToReset" statement in CompositionConn3 in Sec. 4, has three ExpectedMappings clauses, each mapping different application operation(s) to replaced_setOP: set, setX, and setY, respectively. Each of these mappings has its own implementation of reset (usage interface of DataToReset). The commonality of the inner objects for these three clauses, i.e., the implementation of eval to invoke replace_setOp is implemented in the abstract subclass of AutoReset.DataToReset, Point-DataToReset-setOp. The first mappings.put invocation within the constructor CompositionConn3(), associates the method for expected_set in Point to a subtype of Point-DataToReset-setOp that implements the abstract method reset to call expected_set(0, 0) on the application object, host. This corresponds to the mapping clause:

```
Point is DataToReset with {
   {setOp = set;
    reset {
      set(0, 0);}
    }
  ...
}
```

The other two clauses are implemented in a similar way by the other two anonymous inner objects passed as parameters to the following two mappings.put invocations in CompositionConn3(). In response to receiving a notify(thisObject, thisClass, expected_meth, args) event, a connector object will look up its mapping table for an entry for expected_meth. If there is one, it will extract the participant object associated to expected_meth and call eval on it, which in turn will invoke the aspect replacement operation corresponding to expected_meth, passing along thisObject, thisClass, expected_meth, and args. The structural relationships between a point object, the singleton CompositionConn3 object and the DataToReset class as well as the message invocation chain following an invocation of set(3, 5) on a point are shown in Fig. 8.

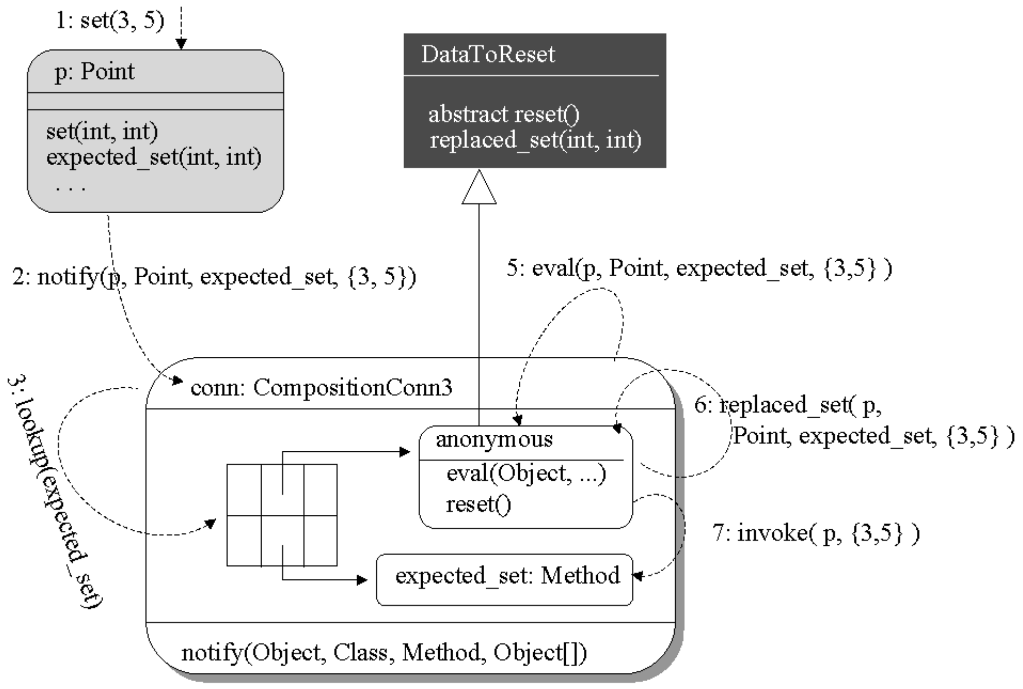

Figure 8: Invoking an Operation on an Adapted Application Class

Aspect deployment can be performed at class and instance level. This can be realized as an option to the connector pre-compiler. If class level deployment is specified, the aspectSubscribers vector will be a class variable, otherwise it will be an instance variable (the code in the class Point above corresponds to the class level deployment). In the case of class level deployment, the compiler will automatically add an instance of the generated connector class to the subscriber set of the application class. Otherwise, this has to be done explicitly in the client code, i.e., the code that uses points.

The approach described so far, requires tool support for binary adaptation as well as code generation. Alternatively, one can workaround the need for binary adaptation support by slightly modifying the code generated for the connector classes. In addition to the definitions above, a connector class will define an adaptation operation, adapt, that expects an object of the application class Point. The method defines an inner subclass of Point, whose implementation corresponds to the result of the binary adaptation as far as operations affected by the aspect are concerned. Application operations that are not affected by the aspect are reimplemented to simply delegate to the original point object passed as a parameter to the adapt operation. An instance of the inner subclass is returned. Deploying aspects with instances of the application is explicitly done by the clients, as illustrated by the sample code below:

```
class Client1 {
  . . .
  public static void main(String[] args) {
     . . .
     Point p = new Point();
     CompositionConn3 connector = CompositionConn3.singleInstance;
     p = connector.adapt(p);
     p.setX(12);
     ....

     p.setX(10); // this is the setX operation number 100

     System.out.println("X coordinate = " + p.getX());
       // will display
       // > X coordinate = 0
       // >

      ...
   }
}
```

One can even go further and workaround the need for code generation by letting the programmer write the code for connector classes instead of using the simple connector language presented in this paper. However, this would be a tedious job for the programmer. Furthermore, we would lose the adaptiveness gained at the connector level, by using special techniques such as pattern matching on operation names and traversal strategies for describing paths (or “long getters”) in the object graph of the application. Further details with regard to implementation strategies and patterns are left out of the scope of this paper.

6 Related work

Adaptive Plug&Play (AP&P) components [22] are the immediate precursors to aspectual components. AP&P components are rooted in Holland’s executable contracts [9] and in Rondo [20]. An AP&P component features already some of the characteristics of an aspectual component, but with different terminology. In AP&P components, the *interface class graph* corresponds to the participant graph of aspectual components. AP&P component instantiation statements correspond to connectors in aspectual components. What is new in aspectual components compared to AP&P components is the **replace** clause that is crucial for expressing “systemic” aspects. The capability to have refinement, not only between components but also between connectors, is also new. In naming our modules “aspectual components” and not just components [19], and in adjusting the terminology used with AP&P components, we want to improve the distinction between components that only offer new functions and components that can also affect other components in a cross-cutting way.

AspectJ from Xerox PARC [27] is also an immediate precursor of aspectual components and has significantly benefited this paper. AspectJ has grown out of the need to make it easy to add more aspects to Java. In her thesis [16] supported by Gregor Kiczales’ team at Xerox PARC, Crista Lopes first implemented the synchronization aspect COOL [18] and then the data transfer aspect RIDL [17], and it was clear that both aspect implementations had something in common that needed to be factored out. In Demeter/Java [14, 4], another implementation of COOL and RIDL, we first developed a generic, but low-level, weaving language, which was used to implement both COOL and RIDL. At the same time, Xerox PARC developed the higher-level weaving language AspectJ that is a definite advance. We noticed the similarities between AspectJ and AP&P components, and the additional leverage AP&P components can offer AspectJ. The main deficiency in AspectJ is that it does not follow the principle of programming against a generic data model that is then separately connected to a concrete data model (or other generic data models). As a consequence, AspectJ aspects are not as reusable as they should be, since they use application level concepts. In contrast, the connectors of aspectual components apply the idea of loosely coupling the behavior aspect to the structure aspect. AspectJ tightly couples structure and behavior, which leads to unnecessarily tangling. Following a modular approach makes aspectual components also easier to understand, reuse and modify than AspectJ aspects.

One motivation behind aspectual components is to capture recurring patterns as aspectual components. This follows the idea that it is useful to turn design patterns into language features, and reclassify patterns on how far they still remain from becoming language features [8]. An example is the Observer pattern (Publisher-Subscriber aspect) shown earlier. Another motivation is the developments in software components. Aspectual components should be components like, e.g., JavaBeans. An example is the use of the Visitor pattern to decouple structure and behavior [?].

Enterprise JavaBeans (EJB) from SUN [26] is a component technology that addresses aspectual decomposition. An enterprise bean provider usually does not program transactions, concurrency, security, distribution and other services into the enterprise beans. Rather, an enterprise bean provider relies on an EJB container provider for these services. EJB defines six distinct roles in the application development and deployment workflow one of which is called *deployer*. A deployer adapts enterprise beans to a specific operational environment. Adaptation is possible because all client requests directed at an EJB object are intercepted by the EJB container to insert lifecycle, transaction, state, security, and persistence rules on all operations based on deployment descriptor settings. Also, customization of the behavior can take place here. Deployment descriptors are defined declaratively at deployment time rather than programmatically at development time.

As an example we consider how persistence is handled by EJB containers. The deployment descriptor of a bean contains an instance variable ContainerManagedFields defining the instance variables that need to be read or written. This will be used to generate the database access code automatically and protects the bean from database specific code.

While EJB makes a good effort to avoid tangling of basic services code with basic bean functionality, it does it in an ad-hoc way. Aspectual components are an attempt to provide a simple linguistic mechanism to control tangling. Persistence can be handled by aspectual components, for example, in the following way. We define an aspectual component with two participants: *source* and *target* that have a zero to many association. *Source* has a method to read and write *target*-objects, namely all *target* objects associated with *source*. A connector is used to specify the mapping of the one edge participant graph onto a complex application class structure. An effective way to specify the connector is to use a traversal strategy.

A propagation pattern [12, 13] is an early precursor of an aspectual component. A propagation pattern is formulated in terms of a generic data model ([13, Fig 4.1 page 79]), that corresponds to a participant graph of an aspectual component. The behavior of a propagation pattern is expressed in terms of the generic data model and wrappers that add code before and after certain methods. The wrappers have the flavor of replace clauses of aspectual components.

Context classes [24] by Seiter et al. have a similar purpose as aspectual components. A method invocation can be modified by a context class by adding code before and after nodes and edges. However, the notions of connectors and a participant graph are absent.

The component-connector terminology of aspectual components is borrowed from software architecture [7].

The Catalysis method [5] has a strong emphasis on modeling collaborations. Collaborations are used to implement connectors between components. In Catalysis, components are pieces of software that can be *plugged into* a wide variety of others.

There are both commonalities and differences between Catalysis collaborations and components and our collaborations and components. In both works, collaborations are handled in a similar way. While

Catalysis uses a *common model of attributes* we use a participant graph. One distinguishing feature is that our components have built-in support to express aspectual decompositions while Catalysis components don't explicitly have this feature. A second distinguishing feature is that the maps between participant graphs and their use in other participant graphs is handled adaptively (using traversal strategies) and safely (using refinement-instance). On the other hand, Catalysis has the advantage that it addresses pre- and post-conditions and invariants.

In the *mixin-layers* approach [25], collaborations are implemented as mixins (outer mixins) that encapsulate other mixins (inner mixins). An outer mixin is called a *mixin layer.* The super-parameter is specified at the level of a mixin-layer (collaboration). By explicitly representing collaborations as mixin layers and by defining the super-parameter at the level of collaborations, Smaragdakis and Batory provide a good technique to programming with behavioral components involving several classes. Aspectual components improve on mixin-layers by covering also aspects and by the connectors to flexibly link components.

Feature models are used in [2] to capture the reusability and configurability aspect of software. Feature models help to separate components and the configuration knowledge for those components. In general, a feature is implemented by a combination of components and aspects. Our work on connectors for aspectual components helps to better express configuration of the aspects and components.

7 Conclusion

The paper makes three contributions to component-based programming:

- We show that programming class collaborations and programming aspects can be done with the same construct: *aspectual components.* We show that the only new feature that needs to be added to *Adaptive Plug&Play* (AP&P) components is a modification interface.
- We show that *Aspect-Oriented Programming* (AOP) is easier if each aspect is programmed against a generic data model, a *participant graph*, which is separately connected to a concrete data model or other generic data models. The usefulness of the separation of components and connectors is well-known in software architecture but has so far not been applied to AOP.
- We show how aspectual components can be implemented in Java.

The term aspectual component, instead of AP&P component, was chosen to send the message that we have a new construct that is useful independently of *Adaptive Programming* (AP). This is not to say that AP is not useful in this context. On the contrary, AP is useful for specifying connectors between aspectual components adaptively.

Future work includes the application of aspectual components as a design tool in a financial software project using Enterprise JavaBeans technology and the development of tools to support development and deployment of aspectual components.

Further information about the paper is available at:

`http://www.ccs.neu.edu/research/demeter/biblio/aspectual-comps.html`

References

[1] Robert Allen and David Garlan. A formal basis for architectural connection. pages 213–249, July 1997.

[2] Krysztof Czarnecki and Ulrich Eisenecker. Synthesizing objects. In Rachid Guerraoui, editor, *European Conference on Object-Oriented Programming*, Lisbon, Portugal, 1999. Springer.

[3] Eric M. Dashofy, Nenad Medvidovic, and Richard N. Taylor. Using off-the-shelf middleware to implement connectors in distributed software architectures. In *ICSE '99. Proceedings of the 1999 international conference on Software engineering*, pages 3–12, 1999.

[4] Joshua Marshall Doug Orleans, Johan Ovlinger, Kedar Patankar, Binoy Samuel, and Karl Lieberherr. Demeter/Java. Technical report, Northeastern University, December 1998. http://www.ccs.neu.edu/research/demeter/DemeterJava/.

[5] D.F. D'Souza and A.C. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[7] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, volume I. World Scientific Publishing Company, 1993.

[8] J. Gil and D. Lorenz. Design patterns and language design. *IEEE Computer Magazine*, pages 118–120, March 1998.

[9] Ian M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.

[10] IBM. VisualAge for java. http://www.software.ibm.com/ad/vajava/.

[11] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *European Conference on Object-Oriented Programming*, pages 220–242. Springer Verlag, 1997.

[12] 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.

[13] Karl J. Lieberherr. *Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns*. PWS Publishing Company, Boston, 1996. 616 pages, ISBN 0-534-94602-X, entire book at www.ccs.neu.edu/research/demeter.

[14] Karl J. Lieberherr and Doug Orleans. Preventive program maintenance in Demeter/Java (research demonstration). In *International Conference on Software Engineering*, pages 604–605, Boston, MA, 1997. ACM Press.

[15] Karl J. Lieberherr and Boaz Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997. http://www.ccs.neu.edu/research/demeter/AP-Library/.

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

[17] Cristina Videira Lopes. Graph-based optimizations for parameter passing in remote invocations. In Luis-Felipe Cabrera and Marvin Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 179–182, Lund, Sweden, August 1995. IEEE, Computer Society Press.

[18] Cristina Videira Lopes and Karl J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In Remo Pareschi and Mario Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.

[19] McIlroy. Mass produced software components. In P. Naur and B. Randell, editors, *Report on a Conference of the NATO Science Committee*, pages 138–150, October 1968.

[20] Mira Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance*. PhD thesis, University of Siegen, 1997.

[21] Mira Mezini and Karl 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.

[22] Mira Mezini and Karl 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.

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

[24] Linda M. Seiter, Jens Palsberg, and Karl J. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.

[25] Yannis Smaragdakis and Don Batory. Implementing layered designs with mixin-layers. In *European Conference on Object-Oriented Programming*. Springer Verlag, 1998.

[26] EJB Team. Enterprise JavaBeans. Technical report, SUN Microsystems, 1999. http://java.sun.com/products/ejb/.

[27] Xerox PARC AspectJ Team. AspectJ. Technical report, Xerox PARC, January 1999. http://www.parc.xerox.com/spl/projects/aop/.