Aspectual Collaborations for Collaboration-Oriented Concerns

KARL LIEBERHERR DAVID H. LORENZ JOHAN OVLINGER

College of Computer Science, Northeastern University,
Boston, Massachusetts 02115-5000, USA
Email: {lieber,lorenz,johan}@ccs.neu.edu

Abstract. Aspect-oriented programming (AOP) controls tangling of concerns by isolating aspects that cross-cut each other into building blocks. Component-based programming supports software development by isolating reusable building blocks that can be assembled and connected in many different ways. We show how these concepts can be integrated by introducing a new component construct for programming called *aspectual collaborations.* We explore how these collaborations can be deployed, composed, and reused.
Aspectual collaborations allow us to capture, in separately compiled units, cross-cutting behavior such as intercessionary advice, exception handling, invariants, and generic behavior. These compiled units can then be flexibly deployed once or several times into base applications or composed with other collaborations. The implementation of a weaver compiler for aspectual collaboration is described.

1 Introduction

Separation of concerns and *modularity* are at the heart of the programming process. Concerns are conceptual units in which one decomposes a given problem. Modules are software units in which one organizes software. Modular programming is conducted by expressing programming concerns in modular units.

The key insight of *Aspect-Oriented Programming* (AOP) [23] is that module boundaries seldom fit along *all* concern boundaries. The modular breakdown captures some programming concerns, while other concerns, namely, crosscutting concerns, are scattered throughout the units of modularity resulting in tangling of the concerns' code. (For a survey of AOP techniques see [10].)

In principle, AOP alleviates the problem of scattering and tangling by separately expressing each crosscutting concern in terms of its own modular structure, as an external aspectual unit, which can then be re-attached, across modular boundaries, to the host application. In practice, however, the state of the art for encapsulating aspectual units leaves much to be desired.

The two leading AOP projects and de facto standards, AspectJ [4] and HyperJ [19], exemplify a tradeoff between flexibility and structure. Following the tradition of meta-object protocols [24] and open implementations [22], AspectJ

Weaver	AspectJ	Aspectual Collaborations	HyperJ
Emphasis	Programming	Components	Software engineering
Approach	Aspectual	Composable	Seperational
Aspect	Aspect class	Collaboration	Hyper-slice
Aspect Hook	Join point	Join graph	Hyper-module
Attachment	Pointcut designator	Graph pattern matching	Hyper-module
Structure	-	+	+
Flexibility	+	+	-
Symmetry	-	+	+

Table 1. The aspectual spectrum

offers programming flexibility with minimal structural constraints. AspectJ discriminates between methods, which capture base behavior, and aspectual units, which affect the base behavior. Aspects can affect the base behavior, however, methods cannot affect the aspectual unit.

HyperJ, on the other hand, emphasizes structured software engineering, even at the expense of programming flexibility. HyperJ [38] is rooted in the subject-oriented programming paradigm [16], which supports the merger and decomposition of separately specified class hierarchies. Unlike AspectJ, HyperJ treats the base and the aspectual behaviors symmetrically, that is, a hyper-slice can model both the base and the aspectual unit.

The two approaches seem to suggest an aspectual spectrum for AOP languages (see Table 1). In this paper, we show that structured units and flexible attachments are not necessarily mutually exclusive properties. We illustrate concretely an aspectual language, in which the aspectual units are both structured and flexible. The novelty of the new aspectual language lies in the following characteristics:

- *Well defined aspectual interface.* The aspectual units emphasize encapsulation above everything else. As a result, separate compilation of aspectual units is made possible.
- *Collaboration-oriented attachments.* Attachments, the equivalent of join points in AspectJ, are collaboration-oriented. That is, *aspectual collaboration* (henceforth AC), the equivalent of hyper-slices in Hyper/J, are parameterized by a formal class graph. Collaborations are attached to host programs using graph pattern matching, which is a generalization of pointcut designators in AspectJ and hyper-modules in HyperJ.

1.1 Well defined aspectual interface

An aspectual interface specifies the contract enforced between the aspectual unit and the host program at every attachment. The aspectual interface controls not only how the host program sees the aspectual unit and how it sees the host program, but also what behavior is expected and what is provided at each attachment occurrence.

The interface must balance the need for well-definedness against the need for flexibility. For example, the trivially complete interface (requiring a copy of the host application) surely allows rich behavior to be woven in, but severely restricts where the aspectual unit can be attached. The trivially empty interface, on the other hand, provides ample opportunities for reuse, but offers no interface for behavior. The interface must thus be structured enough so that interesting aspects can be written against it on the aspectual unit side, yet flexible enough to accommodate a wide variety of host programs on the application side.

An immediate benefit of a well defined aspectual interface is the possibility to *separately compile* aspectual units before composing them. The aspectual interface allows the weaver compiler to analyze aspectual units and the host program in isolation, and then to derive the meaning of the combined whole by additionally analyzing how they have been attached together. This baseline for how much analysis is needed to enable compilation of aspectual interfaces influences where the balance between flexibility and analysis lands.

The difficulty is to retain some measure of generality in the aspectual behavior. For example, an AC for the standard logging aspect should be applicable to methods of any signature, while still have an interface specific enough to enable separate compilation.

1.2 Collaboration-oriented attachments

We take the broad view of considering an aspectual unit to generally be a multi-party behavior involving several roles. We show that AOP is made easier when each aspect is programmed against a generic *Object-Oriented* (OO) model rather than against a single method-like or class-like construct. We generalized the model of join points to include also *join graphs*. An aspect's *formal class graph* is adapted to an *actual class graph*, in the form of a concrete data model or other generic data models. Optionally, traversal-related aspects as well as adapters can be written using *traversal* specifications, which makes the aspects and adapters more *reusable*. A traversal-related concern is a special kind of collaboration-oriented concern where the collaboration consists of traversing through a group of objects and executing code (advice) during the traversal.

1.3 Aspectual Collaborations outlined

An AC has the following ingredients:

- A *skeleton collaboration* as an intrinsically named graph of *participants*, where the nodes are formal classes, and the edges are *is-a* and *has-a* relations and functional relations.
- Required and provided interfaces:
  - A *required interface*, which consists of all expected features, declares holes in the implementation. These need to be provided to complete the collaboration.

  - A *provided interface*, of exported features, which can be exported individually, or in groups (by a participant or a collaboration). The provided interface allows the behaviors of the AC to be accessed from the outside.
- *Aspectual methods*, which are able to intercede in invocations of other methods. These are in general applicable to methods of any signature, but can be constrained to work only with a more limited set of signatures in order to gain functionality.

An AC definition has the following syntactic structure:

collaboration *name*
{ **extends** *collaboration* }*
{ **participant** *formal_class* }*
{ [**match** *roles*] **attach** *collaboration* }*

Outline The outline of the paper is as follows. The next section (Section 2) presents the idea of AC through simple examples, which illustrates the notion of required and provide interfaces. In Section 3, attachments and join graphs are described, delving into details of how state can be shared between deployments (attachments) of the collaborations, and illustrating how to combine collaborations to form larger units. The topic of aspectual methods is explained in Section 4. Section 5 describes the implementation of the AC weaver compiler. Related work is brought in Section 6. Section 7 concludes and describes future work.

2 Aspectual Collaborations by Example

A simple aspectual collaboration, without the optional **extends** and **match–attach** clauses, looks just like a Java package, substituting **collaboration** for **package** and **participant** for **class**. In fact, collaborations are a superset of packages,[1] and a package is accepted as a (grounded) collaboration even without substituting the keywords. Thus, the host package variables (Listing 1.1) is a legal collaboration.

Unlike a package, however, a collaboration is generally a specification with "holes": a participant can reference expected features (attributes and methods) as if they were defined. In adviceSetGetAttribute (Listing 1.2), the HasAttribute participant defines a pair of `set` and `get` methods for an expected attribute, which will be provided only later. The double braces around the method bodies should be read as if they were a single brace. Their sole purpose is to allow our implementation to avoid parsing full Java. The **limited** keyword allows for a "frozen" participant (such as String and `int`), for which all modifications are prohibited, to play the role of AttributeType.

Generally, an AC is a formal class graph (a collaboration), which can be superimposed [20] on another class graph. We refer to the act of superimposing

[1] The association between collaborations and packages is fundamental to our implementation: An AC is separately compiled into `.class` files of a single package.

Listing 1.1. A simple host package

```
package variables;
class Vars {
  String foo;
  Baz bar;
}
class Baz {
  Vars var;
}
```

Listing 1.2. Defining generic setters and getters

```
collaboration adviceSetGetAttribute;
participant HasAttribute {
  expected AttributeType aName;
  public void set(AttributeType aName) {{ this.aName = aName; }}
  public AttributeType get() {{ return aName; }}
}
limited participant AttributeType;
```

as *attaching* a collaboration to a host collaboration; the host collaboration is said to have been *decorated*. Expected features declare holes in the formal class graph, while all other features are introduced by the attachment; and aspectual methods (Section 4) can also intercede in the invocation of methods to which they are attached. Expected and aspectual features offer two alternative ways, explicit and implicit, respectively, to transfer control and information from one collaboration to another.

The adviceSetGetAttribute collaboration (Listing 1.2) defines a formal class graph consisting of two participants, HasAttribute and AttributeType, where the former is expected to have a (direct) reference to the latter. When superimposed on an actual class graph, the adviceSetGetAttribute aspectual collaboration will introduce a setter and a getter for the expected reference. The adviceSetGetFoo collaboration (Listing 1.3), for example, is created by attaching adviceSetGetAttribute to the contents of the variables package to introduce a setter and getter for the `foo` field of Vars.

A **collaboration** definition incrementally builds an AC by first declaring a skeleton collaboration (a set of participants) and then attaching additional collaborations to that skeleton. The skeleton collaboration's participants (a formal class graph) can be directly declared (e.g., HasAttribute and AttributeType in adviceSetGetAttribute) or implicitly acquired from another collaboration using the **extends** keyword (e.g., variables in adviceSetGetFoo). Once built, the skeleton collaboration can be decorated by attaching any number of collaborations to

Listing 1.3. Defining a foo setter getter

```
collaboration adviceSetGetFoo;
extends variables;
attach adviceSetGetAttribute {
  Vars += HasAttribute {
    provide aName with foo;
    export set as set_foo;
    export get as get_foo;
  }
  String += AttributeType;
}
```

it (e.g., adviceSetGetAttribute is superimposed on variables in adviceSetGetFoo). The final formal class graph is the result of the AC declaration.

The += operators in adviceSetGetFoo (Listing 1.3) emphasize that the attached roles HasAttribute and AttributeType may have a structural and functional affect on the local participants Vars and String. Vars is decorated with two methods by the behavior introduced in HasAttribute. In addition to the introduction, the += operator also redirects all references (if any – there are none in this example) from the inserted type, HasAttribute, to the destination type, Vars. String, on the other hand, cannot be decorated at all, but allowed as a *lvalue* of += thanks to the **limited** keyword in adviceSetGetAttribute. However, by redirecting references from AttributeType to String, we achieve type parameterization as a degenerate case of attachment [1, 42].

An AC can be extended and attached to any host collaboration, not just a (grounded) package. When an AC is extended, the host collaboration can reuse its formal class graph as if it was defined locally. When an AC is attached to a host collaboration, the attached formal class graph is mapped against the host formal class graph. Any expected feature in the attached AC needs to be either mapped to a concrete feature, or exported for later mapping. For example, the adviceSetGetTwoAttributes collaboration (Listing 1.4) first extends adviceSetGetAttribute and then attaches the adviceSetGetAttribute collaboration to itself, resulting in a collaboration that introduces setters and getters for *two* attributes.

3 Join Graphs and Multiple Attachments

The detail of how *one* attachment of a collaboration is mapped onto another constitutes a *Join Graph* (JG). A JG is evaluated in the context of both the host collaboration and the attached collaboration. The JG is specified at the *use* of the collaboration, as opposed to the definition, and only references the collaboration's interface. Much of the reuse of aspectual collaborations stems from the decoupling of use from implementation, which is made possible by the rich

Listing 1.4. Attaching a collaboration to itself

```
collaboration adviceSetGetTwoAttributes;
extends adviceSetGetAttribute {
  export AttributeType as AttributeTypeA {
    export set as setA;
    export get as getA;
  }
}
attach adviceSetGetAttribute {
  adviceSetGetTwoAttributes.HasAttribute += HasAttribute {
    export aName as bName;
    export set as setB;
    export get as getB;
  }
  export AttributeType as AttributeTypeB;
}
```

interface of collaborations. JGs can be written manually, as in adviceSetGetFoo, or generated with a matching specification template. In this section, we describe the latter case.

The adviceSetGetFoo collaboration demonstrated attaching adviceSetGetAttribute to introduce a setter and a getter method for the variable `foo`. It would be unsatisfactory, however, to have to write such an attachment for each variable for which a setter and a getter is needed. Using adviceSetGetTwoAttributes instead of adviceSetGetAttribute would cut the effort in half, but won't solve the fundamental problem: the repeated attachments would all look very similar, with the only changes being the type and name of the variable and the class it is defined on.

To this end, ACs offer matching of template attachments with automatic generation of bindings for their variables. A matching specification generates multiple JGs. Listing 1.5 shows a concrete example: the attach clause looks very similar, but with some identifiers replaced by variable references (within < . . . >). Indeed, for the match

$$(\mathsf{HasFields} \mapsto \mathsf{Vars}, \texttt{fieldName} \mapsto \texttt{foo}, \mathsf{FieldType} \mapsto \mathsf{String})$$

the attachment is equivalent to that in Listing 1.2. There are two novelties in listing 1.5: the matching clause and the parameterization over hostcollab.

3.1 Matching

In order to attach getters and setters for some instance variable in the output collaboration, the name of the class it is defined in, its type, and name are required. These data can then be bound to identifiers used in attachment template

Listing 1.5. Defining all setters and getters

```
collaboration addAllSettersGetters(hostcollab);
extends hostcollab;
match {
  role <HasFields> {
    <FieldType> <fieldName>;
  }
} attach adviceSetGetAttribute {
  <HasFields> += HasAttribute {
    provide aName with <fieldName>;
    export set as set_<fieldName>;
    export get as get_<fieldName>;
  }
  <FieldType> += AttributeType;
}
```

to generate a legal attachment. We can generate these bindings by interpreting the **match** clause as a subgraph of the output collaboration, finding all matches.

The constraints in addAllSettersGetters are straight-forward: they match every visible [2] instance variable in every participant of the output collaboration. The keyword **role** is used in place of **participant** to highlight that this is not a declaration of—but rather a pattern to match against—a participant. Each match is applied to the attach template to generate a JG for adviceSetGetAttribute, mapping its participants to participants in the output collaboration.

The **export-as** just renames the new pair of methods: addAllSettersGetters exports the `get` and `set` methods from adviceSetGetAttribute under names that are influenced by the field name they are matched to. Similarly, the actual field (defined on whatever `HasFieldPart` is matched to) is provided to adviceSetGetAttribute to allow the code there to access it. adviceSetGetAttribute in the interface.

The matching clause in Listing 1.5 has nothing but variables, but in general, a matching clause will contain hardwired names as well, which significantly constrain the possible matches. More complicated matches can match every self variable, every pair of getters and setters (illustrated in Section 4.3), or in general any constraint expressible by a subgraph.

3.2 Parameterization

The addAllSettersGetters collaboration also illustrates parameterization. It refers to hostcollab, which is an argument to the collaboration, allowing us to reuse the collaboration by applying it to any other collaboration. By extending the argument, addAllSettersGetters defines a generic mixin-like [37] collaboration.

[2] In this context visibility is a function of **exported** rather than **public** keywords.

Listing 1.6. Attaching all setters and getters

```
collaboration varsgns;
extends addAllSettersGetters(variables);
```

To use the generic getter and setter adder, we need to apply it to the collaboration with the variables in need of getters and setters. Listing 1.6 shows varsgns doing this, by extending the applied addAllSettersGetters. Since addAllSettersGetters extends its argument, we can deduce that varsgns extends variables.

The matching specification in addAllSettersGetters produces the following matching bindings when applied to variables:

$$\left\{\begin{array}{l}(\mathsf{HasFields} \mapsto \mathsf{Vars},\ \texttt{fieldName} \mapsto \texttt{foo},\ \mathsf{FieldType} \mapsto \mathsf{String})\\(\mathsf{HasFields} \mapsto \mathsf{Vars},\ \texttt{fieldName} \mapsto \texttt{bar},\ \mathsf{FieldType} \mapsto \mathsf{Baz})\\(\mathsf{HasFields} \mapsto \mathsf{Baz},\ \texttt{fieldName} \mapsto \texttt{var},\ \mathsf{FieldType} \mapsto \mathsf{Vars})\end{array}\right\}$$

The attachment template is evaluated with each set of the bindings separately, attaching and exporting in Vars the methods `get_bar` and `set_bar` as well as `get_foo` and `set_foo`, and in Baz the methods `set_var` and `get_var`.

4 Aspectual Methods, Sharing, and State

4.1 Aspects with state

Listing 1.7 implements a simple collaboration which maintains some state. Two aspectual methods keep count of how many times `inc` is called between `resets`. We'll use this collaboration to introduce the concepts of aspectual methods and sharing of state between attachments of a collaboration. The collaboration has two commented-out keywords; these are for use in the discussion of sharing, but can be safely ignored for now.

4.2 Aspectual Methods

The novelty of Listing 1.7 are the **aspectual** methods `reset` and `inc`. Unlike **expected** methods and fields which declare explicit holes in the encapsulation interface of a collaboration—where information and control flow can cross collaboration boundaries—**aspectual** methods are *implicit* holes. In the terminology of Filman and Friedman [14], aspectual methods allow the host collaboration to be oblivious to invocation of such aspectual behavior. We prefer to think of aspectual methods as *intercessionary*, as they have the option to intercede in the invocation of advised methods.

Listing 1.7. Count each getter between setters.

```
collaboration counter;
participant Counted {
  /*shared*/ /*static*/ int count;
  aspectual RV_1 reset(HM_1 e) {{
    count = 0;
    return e.invoke();
  }}
  aspectual RV_2 inc(HM_2 e) {{
    count++;
    return e.invoke();
  }}
}
```

Aspectual methods are declared with the keyword **aspectual**, followed by a method signature with one argument and return value whose types are either undefined, or participants defined locally to the collaboration. [3]

Aspectual methods are used by wrapping them around host methods. Although completely possible, they are never called directly. The attachment specification `around foo do bar` sets up aspectual method `bar` to intercept all invocations of the wrapped – or host – method `foo`. The API for the objects representing the wrapped methods (types HM_1 and HM_2 in counter) and return values (RV_1, RV_2) allows `reset` and `inc` to `invoke` the original method, and return the result in a type safe manner without knowing any details about the methods they are wrapping.

Thunks (closure objects) allow separately compiled code to invoke methods with full support for arguments and return values. The aspectual method takes an object representing a thunk of the wrapped method's invocation as an argument, and returns an object representing its return value; these objects are created by code generated by our compiler during the attachment phase (see Section 5 for details and consequences of this approach).

An intercepted invocation opens up a number of options to an aspectual method; it can invoke the host method at any time, multiple times or not at all (returning either a default return value, or perhaps a return value from a previous invocation). Both method thunks and return value objects are plain Java objects, and can be stored in data-structures, passed as arguments, or even persisted to the file system. The aspectual method is able to catch exceptions, and under pain of loss of general applicability, also inspect and modify arguments and returned values.

[3] The signatures for each aspectual method do not absolutely have to be distinct, but for the purposes of this paper, we will assume they are – having the same types for two such methods significantly constrains the legal attachments of the collaboration.

Listing 1.8. Counting all the getters and setters.

```
collaboration usecounter;
extends varsgns;
match {
  role <Part> {
    <FType> get_<name>();
    void set_<name>(...,<FType>,...);
  }
} attach counter {
  <Part> += Counted {
    export count as count_<name>;
    around get_<name> do inc;
    around set_<name> do reset;
  }
}
```

In this case, counter's only participant Counted has two aspectual methods, both referencing the same instance variable count. Both implement *before* advice, by doing their intercessionary behavior before invoking the host method and returning the result.

4.3 A counter example

To illustrate, we present usecounter in Listing 1.8, which extends the collaboration varsgns. Thus, without inspecting the listing, we know usecounter will have the same structure as varsgns, with two participants, three variables, and a getter and setter for each variable. The matching clause of usecounter illustrates hardwiring constraints and multiple occurrences of a variable. The variable <Part> will be matched against all participants that have at least one pair of getter and setter methods, where the getter and setter methods must have similar names *and talk about the same type.* The setter method may have multiple arguments, but the type of the getter method must occur at least once, and the setter must return void. Each such pair will generate a match, which then becomes an attachment of the counter collaboration.

As shown, the counter collaboration is attached three times. Three separate count variables are exported as Vars.count_foo, Vars.count_bar, and Baz.count_vars. These are instance variables, so are separate for each instance of the two classes.

Each time a getter (get_foo, get_bar, get_baz) or setter (set_foo, set_bar, set_baz) is called, the method call is packaged into a thunk and passed to inc (for getters) or reset (for setters). The aspectual method then modifies (incrementing or setting to zero) the appropriate count variable, and then invokes the original method, returning whatever that invocation returns. The generated code unpackages the returned value, and if a getter was called, extracts the contained

Listing 1.9. Matching Doubly linked Participants.

```
collaboration fancyusecounter;
extends varsgns;
match {
  role <Part1> {
    <Part2> get_<name1>();
    void set_<name1>(...,<Part2>,...);
  }
  role <Part2> {
    <Part1> get_<name2>();
    void set_<name2>(...,<Part1>,...);
  }
} attach counter {
  <Part1> += Counted {
    export count as count_<name1>;
    around get_<name1> do inc;
    around set_<name1> do reset;
  }
}
```

result, returning that to the original caller. Thus, the collaboration counts how many times a getter is called between calls to the corresponding setter.

Looking back at the very first mention of variables, we see that Vars and Baz have variables that point to each other. As an aside from our exposition, let us assume that we want to only decorate such pairs of variables with counting, to further illustrate the expressiveness of the match patterns. Listing 1.9 shows a match clause that only matches pairs of getters and setters that implement a doubly linked relationship. [4] The single attachment is not a typo. We only attach to one of the two pairs at a time; the other will be dealt with by symmetry of the match pattern.

4.4 Sharing

The state (the `count`) local to the counter collaboration is *duplicated* for each attachment in usecounter – the collaboration counts calls to `get` between calls to `set` for each object separately. It is natural to ask whether other arrangements are possible – indeed, one can imagine wanting to *share* the count between all attachments to a class or all attachments in one collaboration to any class.

These are all possible by using the **shared** keyword and its close cousin **static** (as commented out in Listing 1.7). The analogy is clear: **static** shares a member between all instance of a class, allowing all instances to access it, but hindering

[4] Somewhat surprisingly, this is also matched by a self-reference. We can introduce mutual-exclusion constraints on variables to avoid that case.

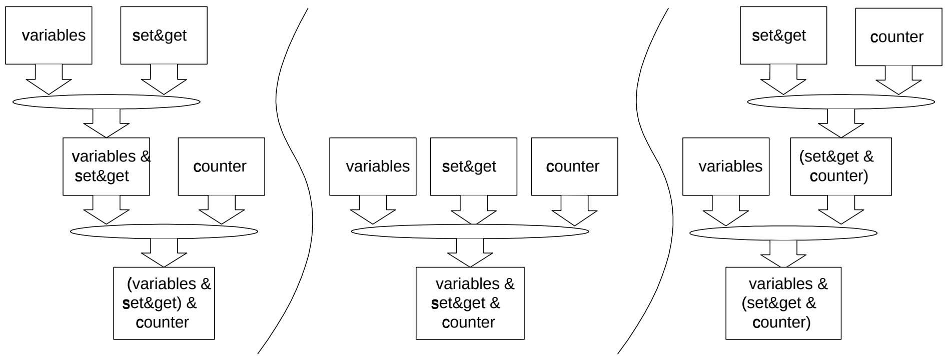

Fig. 1. Sequential attachment (left), simultaneous attachment (middle), and composition then attachment (right) of Collaborations

static members from access per-instance members. Likewise, **shared** members are shared between all attachments of a collaboration, but cannot access per-attachment features.

Were `count` marked **static**, then just like a normal variable, it would be shared between each instance (but it would still be duplicated three times) – the collaboration would count calls to get between calls to set for *any* object, but count the three methods variables' getters and setters separately.

However, were the `count` field **shared** instead, it would be shared among each attachment of the collaboration, so the collaboration would count calls to any of the getters for an object, between any of the setters. Each object would be counted separately; by combining **shared** and **static** we can have one count shared between all objects and all methods.

4.5 Sharing vs. Composition

We cannot really call what usecounter (Listing 1.8) does composition; it builds a collaboration with counted getters and setters by first adding getters and setters to a host with variables, and then adding counters to that. Predictably; we would like to do it the other way around (See Fig 1). In addition to the presentation so far (the left side of the figure), we can also invert the order and first compose counting and getters and setters into one collaboration before adding that to the variables (right side of figure), or just add counting and getters and setters at once (middle).

Listing 1.10 shows the composition of getters and setters (from Listing 1.2) and counters – we assume the counter from Listing 1.7 with the **shared** keyword in the program text (not commented out). The resulting countedGetSet collaboration adds counted getters and setters to any variable; the twist being that we want the counter instance variable to be shared between all attachments of the collaboration. A general issue when mixing sharing and composition is how to

Listing 1.10. Counting all the getters and setters.

```
collaboration countedGetSet;
extends adviceSetGetAttribute;
attach reshared counter {
  HasAttribute += Counted {
    export count;
    around get do inc;
    around set do reset;
  }
}
```

deal with shared features; at some point we will no longer wish to share a shared feature.

Take as an example our counting collaboration; countedGetSet attaches it once to the getters and setters [5]. The attached `count` variable is shared between that one attachment (somewhat trivial sharing).

The question is now whether `count` will be shared when we attach countedGetSet several (three) times to variables to create sharedvarsgns. [6] We can take the argument a step further: if we attach sharedvarsgns to another collaboration several times, will `count` *still* be shared globally, or will the getters and setters of each attachment have a separate one?

We cannot add sharing information to features of a collaboration from the outside, as shared features (like static features) cannot refer to non-shared (non-static) features, and we only know what refers to what at compile time. However, we *can* add sharing to a whole collaboration, or annotate features as shared at the source level, and once annotated, selectively decide when to remove sharing.

A collaboration's sharing annotation doesn't survive a composition, unless kept alive with the **reshared** keyword in the attachment. This is exactly what countedGetSet does; it attaches counter as reshared, so the `count` variable will be shared between all attachments of countedGetSet to a collaboration.

5 Implementation

The prototype compiler for aspectual collaborations follows a fundamental design: collaborations are compiled separately, and composed at the object-code level. The compiler works at the Java byte code level, which turns out to be very convenient since it provides a fully disambiguated version of the collaboration. The format of Java `.class` files lends itself to easy renaming of members, re-targeting references to point to other members, and moving call graphs from

[5] We know trivially that counter is attached exactly once, because countedGetSet has an attach without a matching clause.

[6] To conserve space, we elide showing that composition, as it is identical to Listing 1.5, apart from replacing adviceSetGetAttribute with countedGetSet.

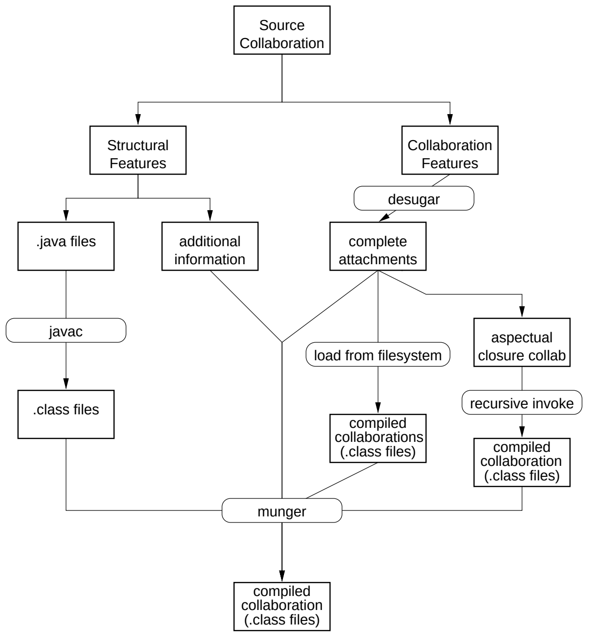

Fig. 2. A data flow over-view of the compiler for Aspectual Collaboration.

one group of classes to another. Figure 2 is a data-flow road-map of how collaborations are compiled to executable programs.

5.1 Compiling collaborations

W.l.o.g., collaborations are assumed to be of a simplified form (without **extend** and **match**). Depending on the form of the collaboration, the participants or attachments may already be simplified. Otherwise, all collaborations are first simplified to a set of participants and a sequence of attachments as follows. The **extends** clause is inlined by cloning the extended collaboration. This is easily achieved by creating empty participants in the same shape as the extended collaboration and attaching it to the newly created participants, exporting every feature. The simplification of the matches to a sequence of attachments is delayed until after the participants have been compiled, to allow matching against bytecodes, which is much more convenient.

For simplified collaborations, participants are compiled by transliterating them to Java and compiling with an off-the-shelf Java compiler. The transliteration replaces **collaboration** with **package**, **participant** with **class**, comments out all additional keywords we have introduced (such as **sharing**, **aspectual**, **expected**), and creating stub bodies for expected methods. After compilation, the `.class` files are annotated with the keywords, so as to ensure that features are treated correctly in subsequent stages. The JVM definition [27] ensures that the annotated bytecodes can still be used on all JVMs (a JVM must ignore all annotations it doesn't understand.) Collaborations can then be unit-tested before composing them by generating stub code for testing. Thus, we leverage the fact that all collaborations are legal packages.

Once the participants have been compiled to `.class` files, the matching specifications (if any) are matched against the compiled classes to generate a sequence of matched name bindings. The attachment template for the matching specification is applied to these name bindings, generating a sequence of complete attachments (*Join Graph* (JG)s). An error is reported if any of the generated attachments is not complete.

5.2 Attachments

The brunt of the implementation effort is the attachment of collaborations according to the complete attachment specifications. An attachment is processed in two stages: first all the participants of the attached collaboration are inserted into the output collaboration, and then each now-decorated output participant is linked according to the attachment specifications. The stages are delineated; the insertion only worries about inter-participant references, while linking is purely intra-participant.

5.3 Insertion

An attached collaboration is physically inserted into the participants of the output collaboration. The $+=$ operators in the attachment specifications specify which participant plays which role, which are interpreted as inserting from the right hand side into the left.

This insertion is achieved by copying the bytecodes from the attached participant into the output participant, under a renaming map. The renaming map consistently renames participant and feature references, so that all features are renamed to unique names, and all participant references that used to reference the attached participants now reference the corresponding output participant.

The name map is what allows **limited** participants to be mapped to non-collaboration types. Limited participants are guaranteed not to need insertion of any bytecodes, so the name-map from the limited participant to whichever participant or class (or even primitive type) that is playing that role suffices to redirect all references from the limited participant to the new type. This incidentally is precisely how one suggestion of parameterized Java [3] works.

Fields and Methods are alpha converted to unique (and unpronounceable) names in order to implement the default action of not exporting attached features and to avoid name clashes with existing features in the output participant. Since we assume that collaborations are closed, we can find all references to a renamed feature and modify those to use the new name. The alpha conversion doesn't affect inter- or intra-participant reference graphs

Some care needs to be taken to not break inheritance and overriding of methods. An overriding method needs to be renamed exactly like the overridden method, in order to work with Java's late method dispatch mechanism. [7] This unfortunately implies that if we override a method inherited from a non-collaboration superclass (for example, `toString` from `Object`) in both the output participant and the attached participant, we are guaranteed to get a name clash. Such cases must be dealt with by the user, as it is seems unlikely that any default heuristic will do the right thing in most cases. Similarly, the instance initialization method `<init>` must have that name if it is to be invoked by the bytecode generated to instantiate a new object. However, the predictability of initialization makes it possible for us to provide sane default behavior for these cases.

5.4 Linking

Once the bytecodes have been inserted into the output collaboration, we need to link up the code from two collaborations, if we wish to ever have control or information flow between them.

Export. The simplest form of linking (and typically performed last, due to interactions with provides) is to export a feature. Even non-hidden features can be exported—in both cases, an export is equivalent to renaming the feature and all references to it.

A small complication is that a feature can only be exported once. This is due to the intrinsic naming of features: since there is only one feature, it can have only one name. With methods, we could work around this by generating forwarding methods, but this approach would not work with fields; thus for regularity, we limit features to being exported at most once per collaboration.

Provide. When we have an expected feature in the same participant as another feature with *exactly* the same signature, we can provide the latter to the former. Providing is implemented as half of an export—all references to the expected feature are redirected to the provided one, but the expected feature is not renamed, but rather removed. Thus the expected and provided feature are now just one—the provided.

The interaction alluded to in exports is that once provided, any export of the expected feature actually exports the provided feature. This may lead to unexpected errors if we also try to export the provided feature – since the two are now just one feature, it can be exported at most once. However, this caveat seems preferable to the alternative of forbidding the export of provided expected features.

Note that the expected and provided features do not need to come from different collaborations—in fact, the only requirement is that they are in the

[7] However, there is no restriction that overloaded methods need to be renamed identically, as overloading is decided statically at compile time.

same output participant and have the same signature. Expected methods can be used to decouple interface and implementation even within a collaboration. Similar arguments apply to aspectual methods.

Around. The most involved linking operation is the wrapping of aspectual methods around host methods. When a host method is wrapped with an aspectual method, we need to proceed in a number of steps:

1. The original host method is renamed to a unique name.
2. depending on the signature of the host method, we generate a HostMethod class to make closures of calls to that method (to hold receiver and arguments), and a ReturnValue class to capture the result (a returned value or exception). The HostMethod class has an `invoke` method that calls the renamed original host method, creating a ReturnValue object from its return value. The names of the generated classes come from the signature of the aspectual method.
3. A new host method is generated to create a HostMethod object with the arguments (if any) to the method, and invoke the aspectual method with this object as an argument. The returned object is necessarily a ReturnValue (as per the signature of the aspectual method) which is unpacked to reveal the wrapped return value, which is returned to the caller of the wrapped method.

The actual implementation uses a template collaboration for the generated code, which is recursively processed by the compiler and inserted and linked into the participants using provide statements.

There is some subtlety involved in aspectual methods; while the names HostMethod and ReturnValue are influenced by the signature of the aspectual method, it is unsafe to have them be those of the original signature—more to the point, they cannot be the same for each attachment. This is obvious after a moment's thought: if we wrap the same aspectual method around two host methods of differing types, we now have two pairs of generated classes with the same name.

It is tempting to keep the signature of the aspectual method the same, and use the declared HostMethod and ReturnValue as common super-types of all attachment generated classes. The returned ReturnValue in step 3 would now have to be downcast to the proper subclass to extract the returned value, but this would seem to work as the HostMethod we instantiate in the same step only returns the expected ReturnValue. Unfortunately, this too is unsafe: since HostMethod and ReturnValue are first class objects, there is nothing to guarantee that the aspectual method plays nice and returns the ReturnValue that came from the thunk it was passed. It could just as well return one from a different attachment—containing a different type result—that it got via a shared global variable. This would result in the downcast of step 3 step failing.

In effect, the HostMethod and ReturnValue are *existential* types; they are different for each occurrence. We implement this by recognizing aspectual methods in the insertion stage and mapping the signature types differently for each attachment.

6 Related work

Adaptive Plug&Play (AP&P) components [32] and the follow-on report [25] are the immediate precursors to aspectual collaborations. AP&P components are rooted in Holland's executable contracts [18] and in Rondo [31]. This work builds on [25], but with significant modifications from experience with implementation, and with a highly modified attachment / matching model. The capability to have refinement, not only between collaborations but also between adapters, is also new. However, we retain the goals and direction of AP&P components. In naming our modules "aspectual collaborations" and not just collaborations [5], we want to improve the distinction between collaborations that only offer new functions and collaborations that can also affect other collaborations in a cross-cutting way.

Mezini and Herrmann [17] discuss a software engineering environment capable of combining dynamic plugability, separate compilation, and aspectual attachment. It is unclear how their PIROL system deals with type safety.

AspectJ from Xerox PARC [40] is also an immediate precursor of aspectual collaborations and has significantly benefited this paper. In her thesis [28] supported by Gregor Kiczales' team at Xerox PARC, Crista Lopes first implemented the synchronization aspect COOL [30] and then the data transfer aspect RIDL [29]. It was clear that both aspect implementations had something in common that needed to be factored out. AspectJ grew out of this attempt, as a [successful] attempt to add general aspects as an integrated language feature to Java. As a consequence of their tight integration with the host program, AspectJ aspects are not as reusable as they could be. By following a modular approach, we hope to make aspectual collaborations easier to understand, reuse and modify than AspectJ aspects.

Multi-dimensional Separation of Concerns and the Hyper/J work [39] generalizes the ideas behind Subject-Oriented Programming [16, 33] by moving to finer grained units of combination. A Hyperslice is a named set of methods and fields in a set of classes. The slice can be added to new classes in a very similar way to collaborations. Similarly to our reuse of a Java compiler for type checking, we could likely have reused Hyper/J for weaving together our collaborations. We chose not to as our needs are very simple, and it seemed an equal amount of work to write our own class munger as to interface with Hyper/J.

Clarke and Walker [7] introduce the concept of composition patterns that is very similar to our aspectual collaborations. However, composition patterns are intended to be used at the design level to model aspects using an extension of UML while our work concentrates on the programming language level. It is also unclear how composition patterns capture multiple attachments. [6] compares how well Hyper/J and Aspect/J can capture composition patterns.

Tarr and Ossher also discuss the need for *sharing* annotations in [34], which this paper addresses.

Context classes [36] by Seiter et al. have a similar purpose as aspectual collaborations. A method invocation can be modified by a context class by adding code before and after nodes and edges. However, the notions of adapters and a participant graph are absent.

The Catalysis method [9] has a strong emphasis on modeling collaborations. There are both commonalities and differences between Catalysis collaborations and our collaborations. In both works, collaborations are handled in a similar way. While Catalysis uses a *common model of attributes* we use a participant graph. One key distinguishing feature is that our collaborations have built-in support to express aspectual decompositions while Catalysis collaborations don't explicitly have this feature.

In the *mixin-layers* approach [37], 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 collaborations involving several classes.

Feature models are used in [8] 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 adapters for aspectual collaborations helps to better express configuration of the aspects and components.

An early paper on binary component adaptation, a technique used in this paper, is [21].

The ABB Aspect architecture [2] is a commercial system that uses AC with only one participant. This shows that even with this significant restriction you can build useful industrial control systems.

Erik Ernst has addressed issues of collaborations that are relevant to our work but which we have not yet integrated. The notion of family polymorphism [12] is useful for AC and as the example in [13] demonstrates, the programming language gbeta can simulate basic AC. However, gbeta does not directly support (yet) composite aspectual collaborations.

7 Conclusion

An AOP problem addressed by several authors, (e.g., [11]) is that aspects may be tightly integrated into other code, and therefore it is difficult to tease out and reuse them. The reason why aspects are tightly integrated with other code is the lack of an interface between the aspectual unit and the rest of the system. In this paper, we address this problem. We show that writing aspects against formal participant graphs, and attaching them to other participant graphs, helps to make the aspects more abstract and reusable.

The paper presents aspectual collaborations, a new module with support for aspectual behavior and separate compilation. An AC comprises aspectual behavior (expressed as Java code with holes) written over a formal class graph;

a way of specifying how such a formal class graph is to be attached to a host class graph; and a matching mechanism for generating such attachments.

We show how the decomposition allows us to implement separate compilation of aspectual and additive behavior; allows composition and parameterization of collaborations; and allows transparently interface with existing Java programs. We describe our prototype implementation of a weaver compiler for aspectual collaborations in Java.

7.1 Future implementation issues

There are several subtle implementation issues that need to be dealt with in future work. We mention the three main ones:

Parameterization. Separately compiling collaborations is desirable. However, parameterized collaborations leaves vital information undeclared until application time. The chains of application can be made arbitrarily long, which makes parameterized collaborations difficult to compile. Currently, the system *is* able to compile all participants directly defined in a parameterized collaboration, but not any attachments or extending other collaborations.

Matching. Somewhat counter-intuitively, adding variables to a matching specification will often increase the number of attachments generated. This is because all distinct matches must be generated, so a specification with fewer variables will have fewer possible distinct matches. Of course, adding constraints without variables to the matching specification will tend to limit the number of matches, as per intuition.

Sharing. The actual implementation of sharing falls slightly behind the design presented. Rather than shared features being shared between all attachments of a collaboration, the system shares them between all attachments with the same participant bindings but possibly different feature bindings. This is a last-minute concession to type safety that was only discovered in the course of implementing the compiler. We have a workaround that reorganizes shared state to a separate class, but it is still unsatisfactory from both an elegance and completeness standpoint.

7.2 Other future work

In addition to those implementation issues, the following are natural extensions to this work:

Adaptive Matching. The matching language incorporates experience gained in adaptive programming [26], supporting matching constraints that restrict roles by reachability over an adaptive path. Our hope is that adaptive matching specifications will allow us to talk about object relationships rather than class relationships. ACs are compile-time entities, providing support for multi-class collaborations. However, at run-time collaborations are typically multi-*object* entities. Adaptive matching specifications can potentially provide support for managing the objects that form the collaboration by generating code to collect objects according to the adaptive matching specification. However, these details are still germinating, and not stable enough for airing in public.

Contract Checking. Design by contact is a well-known approach to designing object-oriented programs. Recently, contract checking at run-time has been reevaluated [15] and we plan to apply this improved contract checking to aspectual collaborations.

Object Graph Constraints. A key concept of collaborations is that each has its own class-graph, which are fused when one is attached to another. The behavior of a class-graph will in general instantiate classes of that class graph and store the objects in variables – in effect, each collaboration will build its own object-graph. In addition to *building* and object graph, the collaboration also makes assumptions about it – these assumptions are encoded in the code of the collaboration, and take the form of invariants over the object-graph.

Examples of invariants are that a non-zero value for one variable indicates that another is ready to be read, or that two variables of the same type in fact *alias* the same object. The key insight here is that the fused collaborations must make compatible assumptions about their object-graphs, as in addition to sharing a fused class-graph after attachment, they will at runtime also share an object-graph.

It would be helpful to capture these constraints in the interface of the collaboration, so as to be able to catch such attachment errors at compile-time. This can be seen as a special case of contract checking, where perhaps machine analysis can help derive the object-graph (run-time) constraints to be checked at compile-time.

Static Analysis and Error Handling. To a large extent, our implementation is designed to leverage the Java type system to provide safety checks – indeed all attachment specification errors will be caught by the JVM before the program is run. However, we would like to be able to catch errors earlier, and to provide the user with more meaningful errors than null pointers and method-not-found.

To this end, we currently verify that participant mappings are consistent with the class-graphs of each collaboration, and that a provided feature matches the signature of the expected feature. However, this only catches errors at attachment time. Long chains of parameterization and collaboration application can delay attachment until long after the compilation of the collaboration where the error will occur.

Since the errors will still be caught at compile-time, catching them as early as possible is an important element of the usability of the compiler for large projects.

Acknowledgment

This research was partially supported by the National Science Foundation under award number CCR-0098643.

References

1. M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, NY, 1996.
2. ABB Team. ABB Aspect architecture – Industrial IT home page. http://www.abb.com/control. Continuously updated.
3. O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* in *Special Issue of SIGPLAN Notices*, volume 32, pages 49–65, 1997.
4. X. P. AspectJ Team. AspectJ home page. http://aspectj.org. Continuously updated.
5. G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Object Technology Series. Addison Wesley, 1999. ISBN 0-201-57168-4.
6. S. Clarke. Designing reusable patterns of cross-cutting behavior with composition patterns. In the Second Workshop on Multi-Dimensional Separation of Concerns in Software Engineering (ICSE 2000), 2000.
7. S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *International Conference on Software Engineering*. ACM Press, 2001.
8. K. Czarnecki and U. Eisenecker. Synthesizing objects. In R. Guerraoui, editor, *European Conference on Object-Oriented Programming*, Lisbon, Portugal, 1999. Springer.
9. D. D'Souza and A. Wills. *Objects, Components, and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
10. T. Elrad, R. Filman, and A. Bader. Aspect-Oriented Programming. *Communications of the ACM*, 44(10):28–97, 2001.
11. E. Ernst. Separation of Concerns and Then What? In *Workshop on Advanced Separation of Concerns, ECOOP*, Cannes, France, 2000. Electronic form: http://www.cs.auc.dk/ eernst/.
12. E. Ernst. Family polymorphism. In *ECOOP*, pages 303–326, 2001.
13. E. Ernst. Loosely coupled class families. 2001. Electronic form, (http://trese.cs.utwente.nl/Workshops/ecoop01asoc/newpage11.htm).
14. R. E. Filman and D. P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA*, Minneapolis, USA, 2000. http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf.
15. R. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *Object-Oriented Programming Systems, Languages and Applications Conference,* in *Special Issue of SIGPLAN Notices*. ACM, October 2001.
16. W. Harrison and H. Ossher. Subject-oriented programming (A critique of pure objects). In *Proceedings OOPSLA '93, ACM SIGPLAN Notices*, pages 411–428, Oct. 1993. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 28, number 10.
17. S. Herrmann and M. Mezini. PIROL: a case study for multidimensional separation of concerns in software engineering environments. In *OOPSLA*, pages 188–207, 2000.
18. I. M. Holland. *The Design and Representation of Object-Oriented Components*. PhD thesis, Northeastern University, 1993.
19. IBM Research Team. Hyper/J home page. http://www.research.ibm.com/hyperspace/HyperJ/HyperJ.htm. Continuously updated.

20. S. Katz. A superimposition control construct for distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(2):337–356, April 1993.
21. R. Keller and U. Hölzle. Binary component adaptation. In *ECOOP'98 - Object-Oriented Programming. 12th European Conference. Proceedings*, pages 307–29, Brussels, Belgium, 20–24 1998. Springer-Verlag.
22. G. Kiczales. Beyond the black box: Open implementation. *IEEE Software*, (1), January 1996.
23. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *European Conference on Object-Oriented Programming*, Budapest, 2001. Springer Verlag.
24. G. Kiczales, J. D. Rivière, and D. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.
25. K. Lieberherr, D. Lorenz, and M. Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999. www.ccs.neu.edu/research/demeter.
26. K. J. Lieberherr and B. 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/.
27. T. Lindholm and F. Yellin. *The Java[tm] Virtual Machine Specification.* Addison-Wesley, 1999.
28. C. I. V. Lopes. *D: A Language Framework for Distributed Programming.* PhD thesis, Northeastern University, 1997. 274 pages.
29. C. V. Lopes. Graph-based optimizations for parameter passing in remote invocations. In L.-F. Cabrera and M. Theimer, editors, *4th International Workshop on Object Orientation in Operating Systems*, pages 179–182, Lund, Sweden, August 1995. IEEE, Computer Society Press.
30. C. V. Lopes and K. J. Lieberherr. Abstracting process-to-function relations in concurrent object-oriented applications. In R. Pareschi and M. Tokoro, editors, *European Conference on Object-Oriented Programming*, pages 81–99, Bologna, Italy, 1994. Springer Verlag, Lecture Notes in Computer Science.
31. M. Mezini. *Variation-Oriented Programming Beyond Classes and Inheritance.* PhD thesis, University of Siegen, 1997.
32. M. Mezini and K. 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.
33. H. Ossher, M. Kaplan, W. Harrison, A. Katz, and V. Kruskal. Subject-oriented composition rules. In *Proceedings OOPSLA '95, ACM SIGPLAN Notices*, pages 235–250, Oct. 1995. Published as Proceedings OOPSLA '93, ACM SIGPLAN Notices, volume 30, number 10.
34. H. Ossher and P. Tarr. Some Micro-Reuse Challenges. In *Workshop on Advanced Separation of Concerns, ECOOP*, Budapest, Hungary, 2001.
35. J. Palsberg, C. Xiao, and K. Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, Mar. 1995.
36. L. M. Seiter, J. Palsberg, and K. J. Lieberherr. Evolution of Object Behavior using Context Relations. *IEEE Transactions on Software Engineering*, 24(1):79–92, January 1998.
37. Y. Smaragdakis and D. Batory. Implementing layered designs with mixin-layers. In *European Conference on Object-Oriented Programming.* Springer Verlag, 1998.

38. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, Los Angeles, 1999. ACM.
39. P. Tarr, H. Ossher, W. Harrison, and S. M. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *International Conference on Software Engineering*, pages 107–119, 1999.
40. X. P. A. Team. AspectJ. Technical report, Xerox PARC, January 1999. http://www.parc.xerox.com/spl/projects/aop/.
41. M. Wand and K. Lieberherr. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University, May 2001.
42. P. Wegner. The object-oriented classification paradigm. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 479–560. Cambridge, Massachusetts, MIT Press, 1988.

A Definitions

We will be using relations to describe graphs and graph matchings that produce join graphs. [8]

If A and B are sets, a relation from A to B is a subset R of $A \times B$. If $a, b \in R$, we will write $R(a,b)$, $a\ R\ b$, and $(a,b) \in R$ interchangeably. A relation from A to A is often called a relation *on* A.

We denote composition of relations by concatenation, e.g., $x\ (RS)\ z$ iff there exists a y such that $x\ R\ y$ and $y\ S\ z$. We also write $x\ R\ y\ S\ z$. R^* denotes the reflexive transitive closure of R.

We often think of directed graphs as relations (and vice versa), so we write $C(c_1, c_2)$ or $c_1\ C\ c_2$ when there is an edge from c_1 to c_2 in C. We take as given the definition of a path in a directed graph.

Definition 1. *A* class graph *consists of a set C (of "classes"), a set E (of field names), for each $e \in E$ a relation (also named e) on classes ("has part named e"), and a reflexive, transitive relation $\leq$ on classes ("is a subclass of"). We write $C(c_1, c_2)$ iff there exists $e \in E$ such that $e(c_1, c_2)$.*

We use C to denote the entire class graph $\langle C, E, \leq \rangle$. We write $\geq$ for the inverse of $\leq$.

An object graph is a model of the objects, represented in the heap or elsewhere, and their references to each other:

Definition 2. *If C is a class graph, then an* object graph *of C consists of:*

1. *a set O (of "objects"),*
2. *a map* $\text{class} : O \to C$, *and*
3. *for each $e \in E$, a relation (also denoted e) on O*

such that if $e(o_1, o_2)$, *then*

$$\text{class}(o_1)\ (\leq e \geq)\ class(o_2)$$

We say that o is of type c *when* $\text{class}(o) \leq c$.

[8] This material is adapted from [41].

As we did for class graphs, we use O to denote the entire object graph whose set of objects is O.

All parts are optional (allowing for null values) or multi-valued (for a given object o_1, there may be many objects o_2 such that $e(o_1, o_2)$). The latter case allows us to handle collections: if class c_1 contains a field e that is a collection of objects of type c_2, we may represent this as $e(c_1, c_2)$ and use multi-valued edges in the object graph, rather than introduce the notion of collections into our model.

In order to define graph cut designators, we use strategy graphs. Several variations of this concept are useful for defining graph patterns. For example, we can put additional constraints on a relation between two states that requires that in the class graph the corresponding path consists of at most one part-of edge.

Definition 3. *A* strategy graph *is given by a set of states Q, a relation S on states, a map* class $: Q \to C$*, a set $QI \subseteq Q$ of initial states, and a set $QF \subseteq Q$ of final states. We denote such a strategy graph by S.*

Definition 4. *A path $p = (o_1, \ldots, o_N)$ in O is an S-*path *iff there is a subsequence $o_{j_1}, \ldots, o_{j_K}$ of p and a path $(q_1, \ldots, q_K)$ in S such that for each i, o_{j_i} has type* class(q_i)*, $j_K = N$, and $q_K \in QF$. As before, we say that an S-path is* minimal *iff it has no initial segment that is also an S-path.*

Paths in object graphs and class graphs are connected as follows. We start with a fixed class graph C.

Lemma 1. *There exists an object graph O of C and objects o_1, o_2 such that $O(o_1, o_2)$ iff* class$(o_1) \leq C \geq$ class(o_2).

Proof. See [41].

Lemma 2. *There exists an object graph O of C and objects o_1, o_2 such that $O^*(o_1, o_2)$ iff $class(o_1)$ $(\leq C \geq)^*$ $class(o_2)$.*

Proof. See [41].

Lemma 3. *Let c_1 and c_2 be classes. Then there exists an object graph O of C and objects o_1, o_2 such that* class$(o_1) \leq c_1$ *and* class$(o_2) \leq c_2$ *and $O^*(o_1, o_2)$ iff $c_1 \geq$* class(o_1) $(\leq C \geq)^*$ class$(o_2) \leq c_2$.

Proof. Immediate.

B Graphcut designators

We are interested in expressing that all object graphs that contain certain patterns have certain behaviors. We formulate the patterns in terms of the class graph of the objects.

A graphcut designator $GCD = (S, Var)$ for a class graph C is a tuple where S is a strategy graph. The nodes of S are mapped to classes of C and the edges specify paths between those classes. Var is a set of variables that assume values of nodes and edge labels in C.

Although the strategy graphs used here are identical to the strategy graphs used in Adaptive Programming, their purpose is different. Here they are not used to define a traversal of objects but to find all occurrences of class graphs (in some bigger class graph) that match the strategy. Each such occurrence is called a join graph. The join graph must satisfy the path constraints expressed in the strategy graph. For each such occurrence, the variables in Var are assigned.

As a simple example, consider the match specification introduced earlier:

```
match {
  role <HasFields> {
      <FieldType> <fieldName>;
  }
}
```

It describes a strategy graph with two states HasFields and FieldType and a transition between those two states that requires one part-of edge called fieldName in the class graph. Var consists of HasFields, FieldType and fieldName.

In future work we will investigate how the various constraints we want to express on strategy graph transitions influence the complexity of matching algorithms.

Strategy graphs are introduced in [26] together with an efficient implementation. A simplified form was already in [35].