Association Aspects

Kouhei Sakurai* Hidehiko Masuhara† Naoyasu Ubayashi‡
Saeko Matsuura* Seiichi Komiya*

*Shibaura Institute of Technology
{sakurai@komiya.ise,matsuura@se,
skomiya@sic}.shibaura-it.ac.jp

†University of Tokyo
masuhara@acm.org

‡Kyushu Institute of Technology
ubayashi@acm.org

ABSTRACT

We propose a linguistic mechanism for AspectJ-like languages that concisely associates aspect instances to object groups. The mechanism, which supports *association aspects*, extends the per-object aspects in AspectJ by allowing an aspect instance to be associated to a group of objects, and by providing a new pointcut primitive to specify aspect instances as execution contexts of advice. With association aspects, we can straightforwardly implement crosscutting concerns that have stateful behavior related to a particular group of objects. The new pointcut primitive can more flexibly specify aspect instances when compared against previous implicit mechanisms. The comparison of execution times between the programs with association aspects and the ones with regular AspectJ aspects revealed that the association aspects exhibited almost equivalent for the medium-sized configurations.

1. INTRODUCTION

In aspect-oriented programming (AOP), an aspect is the unit of modular definitions of crosscutting concerns. Aspects may be provided as a different module system from existing ones (e.g., in AspectJ[10]), or may be defined by using an existing module system (e.g., in Hyper/J[16]). In both cases, an aspect serves as the encapsulation of state and behavior, which are represented by instance variables and advice declarations, respectively, in AspectJ-like languages.

AspectJ-like languages run an advice body *in the context of* an aspect instance, in a similar sense that object-oriented languages run a method body in the context of an object. A problem is how to determine an aspect instance as the context of an advice execution, since aspect instances are not usually obvious during the program execution. AspectJ, for example, offers a few mechanisms[1] to this problem:

- *singleton* aspects create only one aspect instance for each aspect declaration. This type of aspects are useful to implement concerns that have system-wide behaviors.
- *per-object* aspects *associate* a unique aspect instance for each object. When an operation in terms of an object triggers an advice execution, the system automatically looks up the aspect instance associated to the object, and uses the instance as the execution context. This type of aspects are useful to implement concerns that have a unique state for each object.

Those mechanisms are useful to certain kinds of crosscutting concerns, but Sullivan et al. pointed out that they do not straightforwardly support *behavioral relationships*, which are the concerns that integrate the behaviors of collections of objects by extending or modifying their respective behaviors[21]. With above mechanisms, such behavioral relationships are usually implemented by creating a singleton aspect with a table for associating the states unique to object groups. The resulted implementations have to have not only the code for the core behavior but also the code for managing association in a single aspect definition.

Subsequently, Rajan and Sullivan proposed instance-level advising by aspect instances as a solution, as demonstrated in their AOP language Eos[19]. In Eos, the programmer dynamically create an aspect instances to represent behavioral relationships. Each aspect can be associated to the objects in its representing relation. When a method is called during program execution, the advice body is executed in the context of each aspect instance that is associated to the target of the call. As a result, the mechanism can cleanly implement such behavioral relationships. However, the mechanism can still be improved with respect to the following problems: (1) it is not flexible in the selection of aspect instances as it always selects with respect to the target object, and (2) it requires additional language constructs in order to distinguish associated objects of the compatible types.

This paper proposes an alternative mechanism called *association aspects*, which also allows us to associate an aspect instance to a group of objects. The mechanism addresses the abovementioned problems by providing a new pointcut primitive that can more flexibly select aspect instances upon advice execution, and can distinguish associated objects without introducing other language constructs. The

[1]There are also mechanisms based on the control flow, but they are not directly relevant to the topic of the paper.

AOSD 04, March 2004, Lancaster UK.

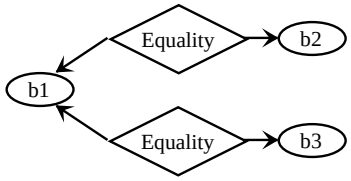

Figure 1: Integration of `Bits`

mechanism is implemented by modifying the AspectJ compiler. Our benchmark tests showed that the association aspects can be implemented with acceptable amounts of overheads in comparison to the singleton or per-object aspects that manually manage tables.

The rest of the paper is organized as follows. Section 2 presents an example of behavioral relationships. Section 3 explains the design of association aspect, our proposed mechanism. Section 4 describes how association aspects are compiled into native Java programs. Section 5 gives the result of our benchmark tests to compare the efficiency of association aspects with respect to the programs in pure AspectJ. Section 6 compares association aspects to similar approaches. Section 7 concludes the paper.

2. MOTIVATING EXAMPLE

This section presents an example system to motivate the need for association aspects. Section 2.1 presents a problem of system integration that becomes a crosscutting concern in object-oriented programming. Section 2.2 then shows that AspectJ implements the concern in an awkward manner. Section 2.3 analyzes the conditions when such problems happen.

The problem presented in this section was first pointed out by Sullivan, Gu and Cai[21]. Readers who are familiar with their work can skip to Section 3.

2.1 System Integration

Integration of independently developed systems often raises crosscutting concerns; it often requires modifications on many descriptions of participating systems[20, 21, 22]. For example, assume that one build an integrated development environment (IDE) system by integrating a text editor and a compiler system[20, 22]. Without AOP, description for the integration concern need to appear in several places in both sub-systems; e.g., a "save" method not only writes to a file, but also needs to invoke the compiler.

For the concreteness, we consider integration of `Bit` objects in the paper. A `Bit` object has a Boolean instance variable and methods for setting, clearing, and getting the value of the variable:

```
class Bit {
    boolean value = false;
    void set()    { value = true; }
    void clear()  { value = false; }
    boolean get() { return value; }
}
```

The integration concern is to synchronize the states of particular `Bit` pairs, which is represented by relations. A relation consists of a type (either equality or trigger) and a pair of `Bit` objects. The relations are created dynamically during program execution.

Figure 1 shows three `Bit` objects (illustrated as ovals) connected by two equality relations (illustrated as diamonds). An equality relation propagates `set` and `get` calls on the left-hand side to the right-hand side and vice versa. Therefore, when `set` is called on b_2, the top equality relation calls `set` on b_1, which in turn makes the bottom equality relation to call `set` on b_3. Note that the relations should not cause an infinite loop; i.e., the call on b_1 by the top equality relation should not be propagated back to b_2.

A trigger relation merely propagates calls on the left-hand side to the right-hand side.

Note: we continue the discussion with a premise that the above system integration is a crosscutting concern, but one might think it could be implemented within traditional object-oriented programming. The interested readers can find detailed discussion on this issue in Sullivan, et al.'s paper [21].

2.2 A Solution in AspectJ

It is possible to define aspects in AspectJ that implement the above relations, but the definitions are not straightforward enough.

Figure 2 shows a possible definition of the equality relation in AspectJ[2]. In order to represent the state of each relation, the aspect defines an inner-class called `Relation`, which has references to the related `Bit` objects and a `busy` flag. The aspect adds a list of `Relation`s to each `Bit` object, so that the advice can find `Relation`s from a `Bit` object.

Two advice declarations capture `set` and `clear` calls, respectively, to any `Bit` object. The bodies of advice obtains a `relations` list from a target object. For each `Relation` in the list, it checks the flag and invokes the same method when the advice is not recursively executed for the same `Relation`.

The static method `associate` creates a relation. When the method is called with two `Bit` objects, it creates a `Relation` object and registers it into each of the `relations` lists in the given `Bit` objects. The integrated system of `Bit`s specified in Figure 1 can be constructed by executing the following code fragment:

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality.associate(b1,b2); //connect b1 and b2
Equality.associate(b1,b3); //connect b1 and b3
```

2.3 Problem of AspectJ Solution

The `Equality` aspect in the AspectJ solution shown in Figure 2 does not straightforwardly model the equality relations.

[2]The definition is written by the authors who follow the outline originally presented by Sullivan, et al.

```
aspect Equality {
  static class Relation {
    Bit left, right;
    boolean busy = false;
    Bit getOpponent(Bit b) {
      return b==left ? right : left;
  } }
  private List Bit.relations = new LinkedList();

  static void associate(Bit left, Bit right) {
    Relation r = new Relation();
    r.left = left;
    r.right = right;
    left.relations.add(r);
    right.relations.add(r);
  }

 after(Bit b): call(void Bit.set())
              && target(b) {
   for (Iterator iter=left.relations.iterator();
        iter.hasNext(); ) {
     Relation r = (Relation) iter.next();
     if (!r.busy) {            //to avoid
       r.busy = true;          //infinite loop
       r.getOpponent(b).set();
       r.busy = false;
     }
   }
 }

 // advice for the clear method goes here
 // ...
}
```

Figure 2: An Implementation of Equality Relation in AspectJ

At design level, an equality relation is an entity that encapsulates the state (related objects and a busy flag) and the behavior (detection and propagation of method calls). It would be straightforward if a relation is modeled by an instance at the programming level. However, the solution models the relation as an aspect declaration (for the behavior) and an instance of an inner-class (for the state).

In addition, the solution has to manage lists of the states for finding associated relations to an object. This is verbose and distracts the programmer's attention from the actual behaviors added by the relation.

In general, aspect instantiation mechanisms in AspectJ are not sufficient to straightforwardly implement concerns that affects a groups of objects, and have stateful behavior. As it is natural idea to encapsulate the state and behavior in an aspect instance, a mechanism that enables to create aspect instances on a per-object-group basis would be useful.

In other words, the *singleton* aspects in AspectJ is not suitable because it can create no more than one instances. As a result, the implementation would have to allocate the states in different objects, and manage a table to keep those objects.

The *per-object* aspects in AspectJ, namely `pertarget` and `perthis` aspects, are not suitable either. This is because only one per-object aspect instance is allowed to exist for each object. In order to represent relations between objects, more than one aspect instances would exist for one object.

We do not believe that this problem is unique to large-scale system integrations. Rather, similar problems could be observed in smaller-scale systems. For example, in the AspectJ implementation of GoF Design Patterns[5] by Hannemann and Kiczales[6], 6 out of 23 patterns manage the relations and their states by using tables.

3. ASSOCIATION ASPECT

3.1 Overview

We propose an extension to the AspectJ's aspect instantiation mechanism, called an *association aspect*, that allows the programmer to associate an aspect instance to a tuple of objects. Association aspects are designed to straightforwardly model crosscutting concerns like behavioral relations, which coordinate behavior among a particular group of objects.

```
aspect Equality perobjects(Bit, Bit) {
  Bit left, right;
  Equality(Bit l, Bit r) {
    associate(l, r);       //establishes
    left = l; right = r;   //association
  }
  after(Bit l) : call(void Bit.set())
      && target(l) && associated(l,*){
    propagateSet(right);   //when left is called,
  }                        //call set on right
  after(Bit r) : call(void Bit.set())
      && target(r) && associated(*,r){
    propagateSet(left);    //when right is called,
  }                        //call set on left
  boolean busy = false;    //indicates if the
                           //relation is active
  void propagateSet(Bit opp) {
    if (!busy) {           //call set on opp
      busy = true;         //unless it already has
      opp.set();           //propagated
      busy = false;
  } }
  // advice decls. for clear method go here
}
```

Figure 3: Equality Relation with Association Aspect

Two basic functions support the association aspects: (1) a function to associate an aspect instance to tuples of objects, and (2) a function to select aspect instances based on the association at advice execution.

Figure 3 shows the `Bit` integration example rewritten with the association aspect. The `perobjects` modifier on the first line declares that its instance is to be associated to a pair of `Bit` objects. The following statements builds the integrated `Bits` in Figure 1:

```
Bit b1 = new Bit(), b2 = new Bit(), b3 = new Bit();
Equality a1 = new Equality(b1,b2);
Equality a2 = new Equality(b1,b3);
```

The `new` expressions create `Equality` aspect instances. The constructor of `Equality` associates the created instance to the given `Bit` objects.

The `associated` pointcuts in the advice declarations specify what aspect instances shall be used as the execution context of the advice bodies. The combination of pointcuts `target(l) && associated(*,l)` selects aspect instances that are associated to the current target object. The selected aspect instances serve as execution context of advice; i.e., the body of advice runs with accesses to the instance variables of the selected aspect instances.

For example, when a program evaluates `b2.set()`, aspect instance `a1` is selected by the second advice, and executes the advice body. The advice checks `flag` in `a1`, and calls `set` on `left`, which is bound to `b1` in `a1`.

We hereafter refer to the process that selects aspect instances and runs advice body in the context of selected instances as *advice dispatching to aspect instances*.

The following subsections explains the association and advice dispatching mechanisms in greater detail.

3.2 Creating and Associating Aspect Instances

Association aspects are declared with `perobjects` modifiers. They are defined by the following syntax:

`aspect` A `perobjects(`T`,...) { `*mdecl* `... }`

where A is the name of the aspect, T is the type of objects to be associated, and *mdecl* is the member declaration including constructor, method, variable, advice, etc.

The association aspects can be instantiated by executing a `new` A`(...)` expression, which is similar to objects. Creation of a new aspect instance also invokes a constructor for initialization. A newly created aspect instance is not associated to any objects.

The `perobjects(`T_1`,`T_2`,...,`T_n`)` modifier automatically defines an `associate` method in A. It takes n objects of type $T_1, \ldots, T_n$, and associates the aspect instance to the given objects $o_1, \ldots, o_n$. The modifier also defines a `void` A`.delete()` method, which revokes association.

In contrast to per-object aspects in AspectJ, creation and association of association aspects are explicit. This is due to the typical usage of association aspects, in which they represent explicit artifacts such as the `Equality` relations in the `Bit` integration example. When association aspects are required for objects in certain join points, it is possible to make those operations non-intrusive by defining advice as we will see in Section 3.4.

3.3 Dispatching to Aspect Instances

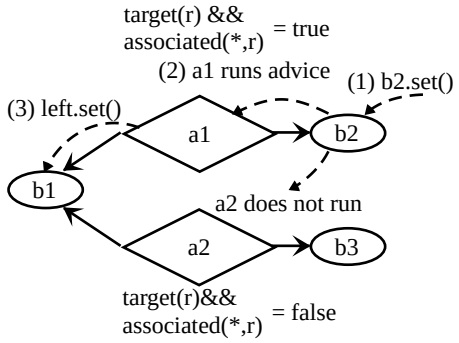

Figure 4: Advice Dispatching to Associated Aspects

Semantically, dispatching advice to aspect instances is realized by trying to execute the same advice in the context of *all* aspect instances, and only the instances that satisfy the pointcut *actually* run the body. In order to select associated aspect instances, we provide a `associated` pointcut primitive.

Figure 4 illustrates the semantics in terms of the example presented at the beginning of the section. The evaluation of `b2.set()` creates a call join point (1). We here focus on the execution of the second advice declaration. Each aspect instance tests the pointcut. Since the pointcut is satisfied only when an aspect instance is associated to `b2` as the second parameter, `a1` is the only aspect instance to run the advice (2). The advice body propagates the call by accessing an instance variable stored in the execution context, `a1` (3).

Aspect instances are ordered in undetermined order to test-and-execute an advice declaration. For around advice, when an aspect instance executes a `proceed` form, the next aspect instance then test-and-executes the same advice.

An `associated` pointcut determines how an aspect instance is associated to objects. In an aspect declared with `perobjects(`T_1`,...,`T_n`)`, the pointcut is written as `associated(`v_1`,...,`v_n`)` where v_i is either

- a variable, which must be bound by another pointcut (e.g, by `target(`v_i`)`), or
- an asterisk (`*`) as a wild card.

An additional restriction is that an `associated` pointcut has at least one bound variable in its parameter.

The pointcut `associated(`v_1`,...,`v_n`)` is evaluated to true for an aspect instance that is associated to $\langle o_1, \ldots, o_n \rangle$, if, for all $1 \leq i \leq n$, v_i is either an asterisk, or v_i is a variable bound to o_i. The asterisks allow more than one aspect instances to match the same join point.

Note that the pointcut distinguishes parameter positions. This is useful to define "directed" relations that captures different events on the different sides of the relations.

3.3.1 Binding to Associated Objects

The `associated` pointcut can bind variables to associated objects when free variables are written instead of wild cards. For example, the following declaration, which is slightly modified from the first advice declaration in the Figure 3, has a free variable `r` instead of the wild card:

```
after(Bit l, Bit r) : call(void Bit.set())
  && target(l) && associated(l,r) {
  propagateSet(r);
}
```

The modified advice has the same behavior to the original one except that it binds `r` to the associated object at the second parameter position when it executes the body.

The binding feature can give shorter definitions to symmetric association aspects, which equally treat their associated objects. For example, the following single advice declaration can substitute for the first two advice declarations in Figure 3:

```
after(Bit b,Bit o): call(void Bit.set())&&target(b)
  && (associated(b,o) || associated(o,b)) {
  propagateSet(o);
}
```

This is because the `associated` pointcuts identify aspect instances that are associated to the target object regardless parameter position, and then the binding feature binds `o` to the associated object that is not the target.

3.4 Static Advice

Association aspects can declare *static* advice, which provides similar semantics to the advice declarations in singleton aspects. When an advice declaration has `static` modifier, pointcut matching and execution is performed exactly once regardless the number of existing aspect instances. Obviously, a static advice declaration may not use `associated` pointcut. The execution context of static advice is the aspect-class; the advice body can only access to static (or class) variables.

The static advice declarations are typically useful for bootstrapping. In order to create a new aspect instance by using the advice mechanism, a static advice declaration should be used because there are no aspect instances at the beginning. For example, the advice in the following code creates an `Equality` instance when `callSomeMethod()` happens:

```
aspect Equality perobjects(Bit, Bit) {
  static after(Bit l, Bit r) :
      callSomeMethod() && args(l,r) {
    new Equality(l,r); //creates an aspect instance
  }
  ...
}
```

3.5 Finding Aspect Instances

It is sometimes necessary to check if there is any aspect instance associated to a particular tuple of objects, or to do something on all aspect instances associated to a particular object (e.g., deleting all aspect instances associated to an object). Those operations can be realized by means of advice declarations with `associated` pointcuts. We therefore do not provide specific primitives for such purposes.

```
aspect Equality perobjects(Bit,Bit) {
 ...
 static void showAll(Bit b) { }    // empty body
 after(Bit b) :
    call(void Equality.showAll(Bit))&&args(Bit b)
    && (associated(b,*) || associated(*,b)) {
  System.out.println(this);  //this is bound to
} }                          //associated instance
```

Figure 5: An Idiom to Enumerate Aspect Instances

An example is to prevent creating no more than one `Equality` aspect instance for the same pair of objects. The next advice does the job:

```
aspect Equality perobjects(Bit,Bit) {
  ...
  Equality around(Bit l, Bit r) :
      call(new Equality(Bit,Bit)) && args(l,r)
      && (associated(l,r) || associated(r,l)) {
    return this;
} }
```

When a program executes `new Equality(`b`,`b'`)` and there is an aspect instance a associated to $\langle b, b' \rangle$ or $\langle b', b \rangle$, the above advice returns a instead of creating new one. When there is no such an aspect instance, a new `Equality` instance will be created because the advice does not run at all.

Enumerating all aspect instances associated to a particular object can be realized by an empty static method with an advice declaration. For example, execution of `Equality.showAll(b)` in Figure 5 displays all aspect instances that are associated to `b`.

4. IMPLEMENTATION

The mechanisms for association aspects are implemented[3] by modifying AspectJ compiler version 1.0.6. Similar to the original compiler, it takes class and aspect declarations as inputs, and generates Java bytecode as compiled code. We first review how the original AspectJ compiler generates compiled code. We then show how the extended compiler generates code for association aspects. For readability, we present compiled code at the Java source-code level.

4.1 Compilation of Regular AspectJ Programs

AspectJ compiler translates an aspect declaration into a class, and an advice body into a method of the class, respectively. Advice is executed by the inserted method calls

[3]The implementation is available at `http://www.komiya.ise.shibaura-it.ac.jp/~sakurai/`. Currently, the `associated` pointcut is provided as a modifier to an advice declaration. This makes the mechanism less expressive, but does not cause a serious problem.

```
class Bit {           //translated
  Counter _aspect;    //associated aspect instance
  boolean value;      //original instance variable
  public synchronized void _bind() {
    if (_aspect == null) _aspect = new Counter();
  }
  //definitions of set, clear and get methods
  //...
}

class Counter {                //translated
  int count = 0;               //instance variable
  public final void _abody0() {//body of the after
    count++;                   //advice
} }
```

Figure 6: Compiled Code by AspectJ

into locations where the pointcut of the advice statically matches. Dynamic conditions in the pointcut (e.g., `cflow` and `if`) are translated into conditional statements inserted at the beginning of translated advice. Masuhara et al. gave a semantic model of the translation by using partial evaluation of an interpreter[11].

Consider the following (non-association) aspect definition which counts invocations of a method on a per-target-object basis:

```
aspect Counter pertarget(callSet()) {
    pointcut callSet() : call(void Bit.set());
    int count = 0;
    after() returning() : callSet() {
        count++;
}   }
```

Compilation of `Counter` aspect with `Bit` class yields the code shown in Figure 6[4]. A statement `b.set();` where `b` is of type `Bit` is translated into the following statements:

```
b._bind();          //create&associate if not yet
b.set();
b._aspect._abody0();//advice dispatching
```

The `Counter` aspect is translated into a class. The variable `count` becomes an instance variable, and the after advice becomes a method.

The `Bit` class has an instance variable `_aspect`, which keeps an aspect instance (i.e., a `Courier` object) associated to the `Bit` object. The `_bind` method creates an associated `Counter` instance for a `Bit` object if it is not yet created.

The translated call to `set` method is surrounded by a call to `_bind` and a call to run the advice body. The latter call is realized by invoking an instance method of `Counter` class. As a result, the body of the advice is executed in the context of an associated aspect instance.

[4]Note that the code is drastically simplified from what the actual compiler generates. For readability, we inlined method calls and renamed compiler-generated methods and fields, and removed unimportant access modifiers.

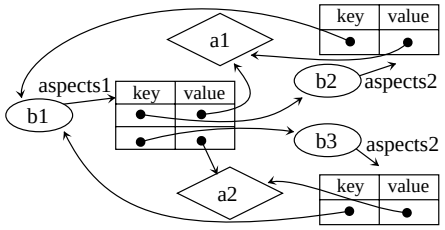

Figure 7: Implementation of Association with Maps

4.2 Compilation of Association Aspects

4.2.1 Compilation of Bit Integration Example

Association aspects are compiled into Java classes in a similar manner to other aspects, except for association and advice dispatching. We first show how `Bit` integration with association aspects is compiled.

The translated `Bit` class has fields `aspects1` and `aspects2` to keep maps from `Bit` to `Equality`:

```
class Bit {// translated
  Map aspect1 = new HashMap(); //Bit -> Equality
  Map aspect2 = new HashMap(); //Bit -> Equality
  ...
}
```

Those two maps are used for processing pointcuts `associated(b,*)` and `associated(*,b)`, respectively. They preserve the following invariants: when an aspect instance a associated to $\langle b_1, b_2 \rangle$, b_1`.aspect1.get(`b_2`)` $= a$ and b_2`.aspect2.get(`b_1`)` $= a$.

Figure 7 shows how the implementation represent the associations of the integrated `Bit`s in Figure 1.

Advice dispatching is translated into a loop over all key-value pairs in a map. A statement `b.set();` is translated into the following code[5] for dispatching the first advice declaration:

```
b.set();                          //original call
for(Bit v: b.aspects1.keys()) { //for the first
   Equality a=aspects1.get(v);  //after-advice
   a._abody0(b);
}
for(Bit v: b.aspects2.keys()) { //for the second
   Equality a=aspects2.get(v);  //after-advice
   a._abody1(b);
}
```

The two for-loops correspond to the two advice declarations. Since the first advice has `associated(l,*)` pointcut where

[5]The syntax `for(`T v `:` e`)` s is a shorthand for looping s for each v of type T in iterator e.

`l` is the target of the call, it processes all the aspect instances `a` in `aspect1` map of the target object, and runs the body of advice by invoking instance method of `a`. The code for the second advice is the same to the first one except for the map.

When all parameters to the `associated` pointcut are bound, advice dispatching is translated into simple look-up in the map. For example, the parameters to the `associated` pointcut in the following advice are both bound by `args`:

```
after(Bit l, Bit r) : call(new Equality(Bit,Bit))
    && args(l,r) && associated(l,r) {
  System.out.println("duplicated!");
}
```

Then the translation of an expression `new Equality(b1,b2)` yields the next statements subsequent to the original expression:

```
Equality a = b1.aspects1.get(b2);
if (a != null) a._abody2(b1,b2);
```

4.2.2 Compilation Strategy

The general compilation strategy is slightly more complicated because we allow to associate aspects to arbitrary number of (i.e., even more than two) objects, and to use wild cards at any parameter positions in `associated` pointcuts.

Basically, the compiler processes `associated` pointcuts for each *binding pattern*. A binding pattern of an `associated` pointcut is a set of all parameter indices where bound variables appear.

Assume aspect A is declared with `perobjects(`T_1`,...,`T_n`)` and pointcut `associated(`v_1`,...,`v_n`)`. Let $bp(v_1,\ldots,v_n) = \{i_1,\ldots,i_k\}$ such that v_j is bound if and only if $j \in bp(v_1,\ldots,v_n)$. Also let $wp(v_1,\ldots,v_n) = \{i_{k+1},\ldots,i_n\}$ such that v_j is a wild card if and only if $j \in wp(v_1,\ldots,v_n)$.

Then, the compiler installs a map into type T_{i_1}. The map is of type $T_{i_2} \rightarrow T_{i_3} \rightarrow \cdots \rightarrow T_{i_n}$ where i_j is j'th index in $bp(v_1,\ldots,v_n)$ or $(j-k)$'th index in $wp(v_1,\ldots,v_n)$. The advice dispatching code is translated, given the values of $v_{i_1},\ldots,v_{i_k}$, into a sequence of map lookup operations followed by $(n-k)$-nested loops that processes over the map.

Below, we present the code for association and advice dispatching when the first k parameters are bound; i.e., $bp(v_1,\ldots,v_n) = \{1,\ldots,k\}$. The code for the other binding patterns is the same modulo permutations of indices.

First, the compiler adds a field `_aspects` of type `Map` into T_1. The initial value of the field is an empty map. The `associate` method is defined like this:

```
void associate(T1 v1,T2 v2,...,Tn vn) {
   Map m1 = v1._aspects;
   Map m2 = getOrCreate(m1,v2);
   Map m3 = getOrCreate(m2,v3);
   ...
   Map mn-1 = getOrCreate(mn-2,vn-1);
   mn-1.put(vn-1,this);
}
```

where `getOrCreate(`m`,`k`)` returns a value for the key k in the map m if it is registered. Otherwise, it creates a new `Map` object, registers it in m with the key k, and returns the created map. We actually use `java.util.HashMap` for a `Map` implementation.

The advice dispatching is realized by inserting calls to the dispatching method `_dispatch` in Figure 8 into the locations where the advice statically matches. The dispatching method takes k parameters from the context (i.e., the join point), finds all aspect instances associated to those values, and calls `_abody` on each found aspect instance.

The `_abody` is the method translated from the advice body, which first checks conditions due to dynamic pointcuts (e.g., `if` and type-tests), followed by the body of the advice.

When the pointcut has all bound variables in its parameter, the dispatching method simply looks up the nested maps until it reaches to an aspect instance.

```
static void _dispatch(T1 v1,...,Tk vk) {
   if(! <parameterless dynamic conditions>)return;
   Map m1=v1._aspects;
   Map m2=m1.get(v2); if(m2==null)return;
   ...
   Map mk=mk-1.get(vk); if(mk==null)return;
   for (Tk+1 vk+1 :  mk.keySet()) {
      Map mk+1 = mk.get(vk+1);
      for (Tk+2 vk+2 :  mk+1.keySet()) {
         ...
            for (Tn vn :  mn-1.keySet()) {
               A a = mn-1.get(vn);
               a._abody(v1,...,vk);
            }
         ...
} } }
void _abody(T1 v1,...,Tk vk) {
   if (! <dynamic conditions>) return;
   //statements in the advice body
}
```

Figure 8: Code for Advice Dispatching and Body

4.2.3 Compilation of Around Advice and Proceed

Compilation of around advice is a little more complicated due to its `proceed` mechanism. In short, our compiler basically follows the strategy in the existing AspectJ compiler, which generates a closure for proceeding.

A difference from the original AspectJ compilation is that the execution of `proceed` in an advice body of association aspect may run the same body in the context of a different aspect instance, whereas the execution of `proceed` in non-association aspect always run a different advice body, or the original join point.

To cope with the difference, our compiler generates a closure that has iterators over maps. When the closure is called, it takes the next aspect instance from the iterators, or it advances to the next advice execution (or the original join point execution) when there are no more aspect instances available in the iterators.

5. PERFORMANCE EVALUATION

We carried out micro-benchmark tests for comparing runtime efficiency between (1) programs with association aspects, (2) programs with singleton aspects that manually manage associated states, and (3) programs with per-object aspects in AspectJ.

All benchmark tests were executed by Sun HotSpot Client Java VM version 1.4.2 beta, running on a Celeron 800MHz Windows XP Professional machine with 512MB memory. Each execution time was measured by averaging the execution time, which is obtained through `currentTimeMillis`, of a loop that runs more than one second.

5.1 Performance of Basic Operations

We measured the costs of the basic operations, namely object creation, aspect instantiation and association, and method invocation with advice execution. They are measured by executing programs that perform the following operations:

1. (**OBJ:**) create n objects that have an n-ary empty method and instance variables,
2. (**ASSOC:**) create an aspect instance and associate it to the n objects, and
3. (**ADV:**) invoke the empty method on an object.

where the aspect has an advice declaration that picks m ($1 \leq m \leq n$) objects from the arguments at the step 3, and finds the aspect instance by using those objects. The body of the advice simply increments five integer instance variables. The aspect definition looks like this:

```
aspect Test perobjects(C,...,C) {
   int x1, x2, x3, x4, x5;
   Test(C o1,...,C on) {
      associate(o1,...,on);
   }
   before() :  callEmptyMethod()
         && args(o1,...,om,*,...,*)
         && associated(o1,...,om,*,...,*) {
      x1++; x2++; x3++; x4++; x5++;
} }
```

There are three actual implementations of the aspect:

AA: that uses an association aspect (shown above),

SNG: that uses a singleton aspect in AspectJ with inner-class objects stored in `HashMap`s for associated states, and

PO: that uses per-object aspect in AspectJ (namely `pertarget`). This is used only when n equals to one.

	n	m	AA	SNG	PO	AA/SNG
OBJ	1		0.951	0.891	0.901	1.07
	2		1.35	1.33		1.02
	3		1.36	1.32		1.03
ASSOC	1		0.320	2.15	0.373	0.15
	2		4.21	7.42		0.57
	3		11.5	16.1		0.71
ADV	1	1	0.0781	0.144	0.137	0.54
	2	1	0.840	0.831		1.01
	2	2	0.194	0.250		0.78
	3	1	1.81	1.59		1.14
	3	2	0.844	0.941		0.90
	3	3	0.310	0.360		0.86

Table 1: Execution Times (in μsec.) of Basic Operations

n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$	n	AA	SNG	$\frac{\text{AA}}{\text{SNG}}$
10	1.32	0.760	1.7	60	9.95	13.7	0.7
20	1.98	1.32	1.5	70	16.1	15.6	1.0
30	3.00	2.20	1.4	80	24.5	24.7	1.0
40	4.50	3.31	1.4	90	53.6	43.1	1.2
50	6.31	12.9	0.5	100	356	161	2.2

Table 2: Execution Times (in msec.) of Bit Integration with n Relations

Table 1 shows the execution times of those basic three operations for different n, m combinations. OBJ denotes the time for generating one object. OBJ and ASSOC times are insensitive to m. The rightmost column shows the relative execution times of AA with respect to SNG.

As we can see, AA poses at most 14% overheads compared to the manual implementation, SNG. Those numbers are reasonable consequence as compiled code for AA basically does the same operations to what SNG does, yet in much more concise descriptions.

5.2 Performance of Bit Integration

We also compared the performance by running the `Bit` integration example in AA and SNG implementations (as shown in Figure 2 and 3, respectively). The benchmark programs first create 100 `Bit` objects, which are randomly connected via n equality and trigger relations, and then invoke `set` and `clear` methods on randomly selected objects for 1000 times.

The overall execution times are shown in Table 2 and Figure 9. As seen in the rightmost column on the table, the relative execution times of AA with respect to SNG range 0.5 to 2.2, depending on the density of the relations. We presume that the difference is caused by the different collection libraries used in AA and SNG. AA uses hash tables for association management in its implementation, whereas SNG uses linked lists. As a hash table usually optimized for medium-sized entries, it could give worse performance for smaller and larger number of relations.

6. DISCUSSION

6.1 Comparison with Eos

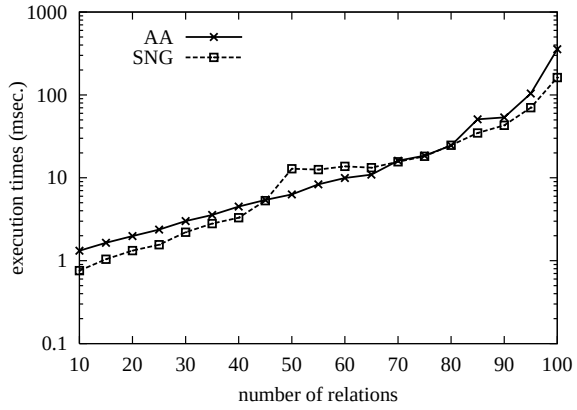

Figure 9: Execution Times (in mesc.) of Bit Integration

As the work on the association aspects is based on the work on Eos[19], we here discuss the differences in detail.

The most notable difference is that Eos implicitly uses the current target object when selecting aspect instances at advice execution. In contrast, association aspects can use arbitrary objects that are explicitly specified by pointcuts. The mechanism in Eos is less flexible for the following situations: (1) when aspect instances should be selected by using a non-target object; e.g., when advising a call to a class method, and (2) when aspect instances should be selected by using more than one object; e.g., when a security concern is to prevent method calls from object A to B, it can be realized by an aspect instance associated to A and B. When a call from A to B happens, all the aspect instances associated to B run an advice body in Eos, even though the caller object A could be used for selecting aspect instances.

Both association aspects and Eos can distinguish roles of associated objects. Eos, however, distinguishes by introducing additional role constructs around advice declarations, which might make it difficult to reuse aspects. For example, even though `Trigger` and `Equality` aspects in Section 2.1 only differ in what objects should be used at advice dispatching, the declarations in Eos have different program structures as the former has to enclose advice declarations in a role construct. Since association aspects distinguish roles of objects by the parameter positions in the `associated` pointcuts, the declarations of those aspects can only differ in the pointcuts. Our approach, in which advice dispatching is govern by pointcuts, would fit the other language features in AspectJ, as it usually reuses aspects through the abstraction mechanisms of pointcuts (i.e., the named pointcuts and the abstract pointcuts).

It may first seem that the implicit dispatching mechanism is more convenient than explicit ones for symmetric associations, but it is actually as not as we first thought. For example, the first two advice declarations in Figure 3 can be written as below with implicit mechanisms:

```
after(Bit b): call(void Bit.set())&&target(b) {
  if (!busy) {
    busy = true;
    if (b == left) left.set(); else right.set();
    busy = false;
} }
```

Although the advice has a shorter pointcut, the body has to check which `left` or `right` is the target in order to propagate the call to the counterpart of the target. In association aspects, it is possible to determine the counterpart of the target by merely using pointcut, as we have seen in Section 3.3.1, which results in simpler advice body.

Both Eos and association aspects should be careful about performance penalty for the objects with no associated aspect instance. For the `Bit` integration example, a `set` call to a `Bit` object that has no associated `Equality` instances should not have significant overhead. There are two possible dimensions to the overhead.

The first is the number of aspect instances. A naive implementation (which is called the first work-around[19]) would significantly degrade its performance to look up a system-wide table of aspect instances. Both Eos and association aspects avoid this problem by having a list of associated aspects in each object.

The second is the number of advice declarations that statically match to the call. Association aspects would linearly degrade the performance as each advice declaration adds a loop over an empty list into the method call expression. Eos avoids this problem by having a list of thunks for each method call expression. However, the approach in Eos requires more memory and more operations for associating/unassociating aspect instances, further investigations are needed to evaluate the tradeoffs.

6.2 Other related work

There was also a similar mechanism that associates aspect instances to objects in the older versions of AspectJ[6].

Mezini and Ostermann propose Caesar, a model for AOP[13, 14]. In Caesar, wrapper instances roughly correspond to aspect instances in AspectJ, but they can be manually instantiated and associated to objects. In addition, the wrapper recycling mechanism helps to retrieve associated wrappers from objects. However, as far as the authors understand, each wrapper can only be associated to one object. The association aspects integrate those instantiation and association-to-objects features into AspectJ's aspects.

Several AOP systems with dynamic weaving mechanisms, such as PROSE[17, 18], Handi-Wrap[2], JBoss[4] and AspectWerkz[3], can create multiple aspect instances from an aspect declaration. If those mechanisms could install aspect instances to a group of objects, behavioral relationships could be straightforwardly implemented. However, as far as the authors understand, those mechanisms install aspect instances on a per-runtime-system basis; i.e., installation of an aspect instance affects all objects belonging to the af-

[6]The versions prior to 1.0 of AspectJ are no longer available publicly. Several literatures[7, 8] that use versions up to 0.6 mention such a feature.

fected classes. Handi-Wrap can define a library to establish association between objects and wrapper instances, though.

Even if those mechanisms could implement what association aspects can do, we believe that association aspects would give more declarative definitions in many applications. With dynamic weaving mechanisms, association and advice dispatching should be understood by means of the side-effects of weaving.

7. CONCLUSION

We proposed the association aspects, which are simple extensions to the aspect instantiation mechanisms in AspectJ. The pointcut-based advice dispatching mechanisms enable flexible yet concise descriptions of aspects whose instances are associated to more than one objects. As a result, the association aspects can give straightforward representations of crosscutting concerns that have stateful behavior with respect to particular group of objects.

We developed a compiler for association aspects by modifying the AspectJ compiler. The benchmark tests exhibited the slowdown factors of the programs using association aspects with respect to the regular AspectJ programs are 0.5 to 2.2. We believe further optimizations are possible to reduce the overheads. For example, eliminating maps that manage associations could improve memory performance.

The future work is to apply association aspects to crosscutting concerns in practical applications. Such concerns would include GUI and security. We are currently examining a GUI application multiple views without changing the original code, and GoF's Design Patterns.

To investigate what design level concepts are appropriate to be modeled by association aspects is also left for the future work. We presume there are many possibilities in concepts like relation objects in UML and roles in collaboration designs. For example, among role model implementations in AspectJ[7], those that are involved with many intrinsic instances could be implemented with association aspects. This would help designing aspect instances in association aspects as well as Eos, which has role constructs to distinguish associated objects.

8. ACKNOWLEDGMENTS

We are very grateful to Kevin Sullivan and Hridesh Rajan for the detailed information on Eos and the comments on the paper. We also would like to thank Tetsuo Tamai, Tomoyuki Kaneko, Etsuya Shibayama, anonymous reviewers and the members of the PoPL and Kumiki meetings at University of Tokyo for their comments on the early draft of the paper.

9. REFERENCES

[1] M. Akşit, ed. *Proc. of AOSD'03*. 2003.

[2] J. Baker and W. Hsieh. Runtime aspect weaving through metaprogramming. In [9], pp.86–98.

[3] J. Bonér and A. Vasseur. AspectWerkz. `http://aspectwerkz.codehaus.org/`.

[4] B. Burke and A. Brok. Aspect-oriented programming and JBoss. O'Reilly Network, May 2003. `http://www.oreillynet.com/pub/a/onjava/2003/05/28/aop_jboss.html`.

[5] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[6] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In [12], pp.161–173.

[7] E. A. Kendall. Role model designs and implementations with aspect-oriented programming. In [15], pp.353–369.

[8] M. Kersten and G. C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In [15], pp.340–352.

[9] G. Kiczales, ed. *Proc. of AOSD'02*. 2002.

[10] G. Kiczales, et al. An overview of AspectJ. In *ECOOP 2001*, pp.327–353, 2001.

[11] H. Masuhara, G. Kiczales, and C. Dutchyn. A compilation and optimization model for aspect-oriented programs. In *Proc. of Compiler Construction (CC2003)*, pp.46–60, 2003.

[12] S. Matsuoka, ed. *Proc. of OOPSLA2002*. Nov. 2002.

[13] M. Mezini and K. Ostermann. Integrating independent components with on-demand remodularization. In [12], pp.52–67.

[14] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In [1].

[15] L. M. Northrop, ed. *Proc. of OOPSLA'99*, Oct. 1999.

[16] H. Ossher and P. Tarr. Multi-dimensional separation of concerns and the hyperspace approach. In *Proc. of the Symposium on Software Architectures and Component Technology*. 2000.

[17] A. Popovici, G. Alonso, and T. Gross. Just-in-time aspects: efficient dynamic weaving for Java. In [1], pp.100–109.

[18] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In [9], pp.141–147.

[19] H. Rajan and K. Sullivan. Eos: Instance-level aspects for integrated system design. In *Proc. of ESEC/FSE*, pp.297–306, 2003.

[20] K. Sullivan. Mediators: Easing the design and evolution of integrated systems. PhD Thesis, Dept. of Computer Science, University of Washington, published as TR UW-CSE-94-08-01, 1994.

[21] K. Sullivan, L. Gu, and Y. Cai. Non-modularity in aspect-oriented languages. In [9], pp.19–27, 2002

[22] K. Sullivan and D. Notkin. Reconciling environment integration and software evolution. *ACM TOSEM,*, 1(3):229–268, July 1992.