Technical Report

Department of Computer Science
and Engineering
University of Minnesota
4-192 EECS Building
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 05-012

Aspects for Modularizing Collaborating Behaviors in Method Implementation

Donglin Liang and Yi Guo

April 06, 2005

Aspects for Modularizing Collaborating Behaviors in Method Implementation

Donglin Liang and Yi Guo
University of Minnesota
Minneapolis, MN 55455, USA
{dliang,yguo}@cs.umn.edu

ABSTRACT

A method of a class in an object-oriented (OO) program often must implement collaborating behaviors that require multiple inter-dependent tasks to be performed within the same period of time to accomplish a certain goal. To achieve the separation of concerns in the description of a collaborating behavior, the tasks involved and the protocol through which these tasks interact should be modularized and described separately. This paper presents a new programming technique to support such a modularization strategy during method design. With examples, we illustrate that our technique may allow the achievement of better modularity in an OO program than using existing techniques.

1. INTRODUCTION

Methods of a class in an object-oriented (OO) program are used to implement the responsibilities that must be provided by the objects of this class [20]. To implement such responsibilities, a method often must perform several inter-dependent tasks, each of which requires the performance of a sequence of actions on appropriate objects. The actions performed for different tasks must be coordinated according to a specific protocol to achieve the desired goal. We refer to this type of behavior that a method must implement as a *collaborating behavior*.

The method call is the only mechanism available in existing OO programming techniques for coordinating actions performed on different objects. With this mechanism, the coordination of the actions must be specified with method call statements that are distributed among the methods that capture these cactions. For example, to ensure that an action captured in method `mA()` of an object of class `A` will be performed immediately after an action captured in method `mB()` of an object of class `B`, a call statement to `A.mA()` may need to be specified in `B.mB()`.

Coordinating actions with method calls may result in undesired code structure when implementing collaborating behaviors in the methods of classes. First, the call statements responsible for coordinating the actions for different tasks in a collaborating behavior are now *scattered* and *tangled* with other statements in the methods of several classes. Therefore, it is difficult to reason about or modify the coordination protocol for these actions. Second, adding call statements for coordination purposes into a method may often introduce responsibilities that are *external* and *unessential* to the object that contains this method. This arrangement reduces the cohesion for this object and increases the coupling between the object and the rest of the system.

Behavior advising has been introduced in aspect-oriented programming (AOP) techniques (e.g., [2, 16, 17]) as a mechanism for coordinating actions performed by an *aspect* instance and other program components. An aspect is a class-like structure.[1] Behavior advising allows the actions (called *advices*) defined in an aspect to be executed in response to its observation of the behaviors of other components whose specifications do not have any mention of the interactions between the aspect and these components. Several kinds of aspect constructs have been developed. The aspect constructs provided by AspectJ [2] can be used to prevent code scattering and tangling when implementing *crosscutting* concerns—"concerns that inherently crosscut the natural modularity of the rest of implementation" [13]. The aspect constructs provided by EOS [16] and EOS-U [17] can be use to prevent code scattering and tangling when implementing behavioral relationship for component integration. Experiences (e.g., [5, 12]) show that appropriate use of these aspects can improve the modularity of the program design.

In an earlier paper [14], we proposed to use aspects and behavior advising to support an *aspectual behavior decomposition* strategy for implementing collaborating behaviors. In this strategy, software developers identify a task from a collaborating behavior, capture this task with an aspect, and use the behavior advising to coordinate the actions of this task with the actions of the other tasks in the collaborating behavior. With the support of appropriate aspect constructs, this implementation strategy can prevent the code scattering and tangling problem in specifying the coordinations between the tasks in the collaborating behavior.

There are two major limitations with the existing aspect constructs in supporting the aspectual behavior decomposition strategy in method implementation. The first limitation is that existing aspects are not suitable for specifying the tasks that a method must closely control for implementing its responsibilities. In languages like AspectJ, a method

[1] An aspect may modify the structures of the classes. In this paper, we focus on the behavior advising only.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 200X ACM X-XXXXX-XX-X/XX/XX ...$5.00.

has no control over what aspects will be used in the program. Thus, the method cannot count on an aspect to provide a task that the method must perform. In EOS and EOS-U, a method can explicitly create instances from aspects and connect, through a special association, a created instance with other instances whose behaviors will be advised. Thus, a method may use an aspect instance to implement a specific task in a collaborating behavior. However, because an aspect is defined as a module, using this approach forces the tasks for a collaborating behavior to be specified in separate modules. In addition, the unified approach for using aspect instances and class instances and the unrestricted manipulation of the associations that control the behavior advising makes it difficult for determining the dynamic impact of an aspect instance on the rest of the program, especially in the presence of aliasing and dynamically-allocated instances. Second, the existing behavior advising mechanism is an intrusive mechanism. This mechanism allows an aspect to intercept, modify, and even substitute the actions performed by other program components. Unrestricted use of this mechanism in a program may complicate the reasoning of the program because a whole program analysis may be required for understanding the meaning of a particular method invocation [4, 3].

This paper presents a new AOP technique that addresses these two problems to better support the aspectual behavior decomposition strategy for implementing collaborating behaviors. Our technique introduces statement-level aspects. A *statement-level* aspect is specified as a subcomponent of an *observation* statement. Through behavior advising, this aspect can coordinate its actions with the actions performed within another subcomponent of this observation statement. Our technique also introduces a thread-based runtime model for the aspects to provide a non-intrusive model for behavior advising. In this model, an aspect and the behaviors that it observes will be run on two separate threads; the behavior advising now becomes a mechanism for coordinating the actions performed by these two threads. In this way, the thread that contains an aspect is not allowed to directly affect the control flow of the thread that contains the behaviors that the aspect observes. Therefore, reasoning about the interactions between an aspect and the rest of the program becomes easier.

A major benefit of our technique is that it provides the first statement-level aspect construct that can be used to form an effective strategy for implementing collaborating behaviors in class methods. An aspect specifies both the actions for a task and the coordination protocol between this task and the other tasks in a collaborating behavior; the other tasks can be specified without including statements for coordination purposes. This strategy achieves separation of concerns in specifying such a behavior.

The statement-level aspects provide at least two benefits over existing aspects in supporting the implementation of collaborating behaviors in class methods. First, a statement-level aspect is specified as an integral part of a method. Like sub-components in other composite statements, the aspect specified in an observation statement can directly access the local variables of the containing method to share data with other tasks; the aspect can also terminate the containing method by executing a *return* statement or raising an exception. Thus, a statement-level aspect may be more convenient to use for specifying a task that the method must perform. Second, the impact of a statement-level aspect on the other program components is well-controlled. A statement-level aspect can be in effect only during the period of time when the containing observation statement is being executed. With the thread-based runtime model, the interactions between the aspect and the other components are quite limited. Therefore, reasoning about the other components will not require the detailed knowledge of the aspect.

The major contribution of this paper is that it presents the core concepts for the statement-level aspects and the thread-based runtime model. Section two presents a motivating example and Section three discusses the major concepts. Section four presents the details of the runtime model. Section five gives an overview of our approach for implementing the aspectual threads. Section six discusses related work. Section seven concludes and discusses future work.

2. A MOTIVATING EXAMPLE

This section uses an example program to illustrate the structure problems that may exist when the actions in collaborating behaviors are coordinated with method calls; the section also presents our solution to such problems. The example program contains an algorithm that propagates entities of type `E` from the root of a tree to other nodes in the tree. This algorithm is a skeleton of the one implemented in an alias analysis framework for Java programs. The program excerpt in Figure 1(a) sketches a Java implementation for this algorithm. In this implementation, the nodes of the tree are instances of class `Node` and the propagation algorithm is implemented in method `prop()`.

Method `prop()` uses a worklist to help propagating the entities to the nodes. In general, when a new entity is added to a node, additional entities may be created; these created entities need to be propagated to the children of this node (described in `Node.process()`), and thus, would require these children to be processed in a similar way. Thus, during the progress of the algorithm, whenever a new entity is being added to a node other than the root, this node must be added to the worklist so that it will be further processed by `Node.process()`. This is guaranteed in `Node.addE()`. Note that `WList.add()` may throw checked exceptions of type `WListExcpt`. This requires both `Node.process()` and `Node.addE()` to declare the throwing of such exceptions. Also note that `Node.process()` and `Node.addE()` may also be invoked by methods that are not shown here.

In this example, maintaining the worklist and processing the nodes can be viewed as two collaborating tasks whose actions must be coordinated appropriately. Figure 1(a) shows that the statements required for maintaining the worklist are now scattered in `prop()` and `Node.addE()`. To allow the worklist to be updated in `Node.addE()`, additional parameters are introduced in various methods to pass a reference of the worklist down the call chain from `prop()` to `Node.addE()`. When `Node.addE()` is called under a context in which the worklist should not be updated (e.g., the first statement in `prop()`), `null` value will be passed into the method. This would require an `if` statement to be introduced in `Node.addE()` for checking such a case. With this implementation, class `Node` is highly coupled with the design for the worklist, which is external to the major purpose of `Node`. Later, if the software developers want to change the type of the worklist or the information to be added to worklist, they will need to change several methods in `Node`.

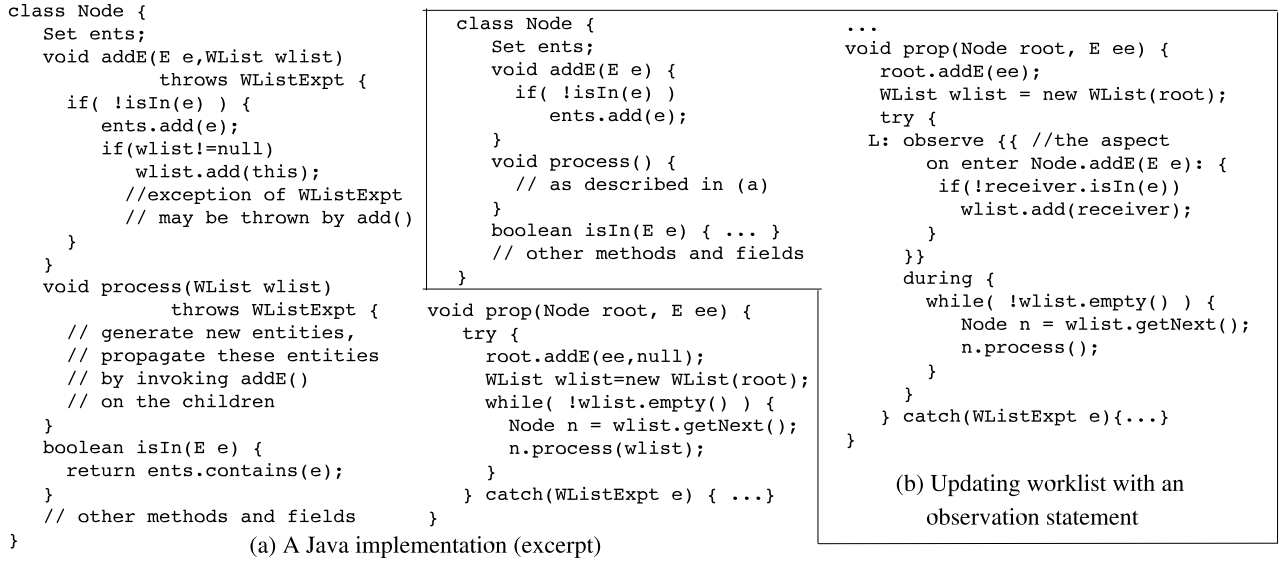

Figure 1: The implementation of a worklist algorithm.

Our solution. Figure 1(b) shows an alternative implementation for the worklist algorithm. This implementation uses an observation statement that we will discuss in this paper for capturing the collaborating behavior for maintaining the worklist and processing the nodes. In the observation statement (the statement starting with keyword `observe`), the node processing is captured in the Java block following the keyword `during`; and the worklist updating is captured in the statement-level aspect defined in the block between keywords `observe` and `during`. The observation statement states that, during the node processing, whenever `Node.addE()` is invoked to add an entity to a node, if the entity is not already associated with the node, then this node must be added into the worklist for further processing.

By comparing this implementation with that shown in Figure 1(a), we can see that, using the observation statement, we can completely capture in `prop()` all the statements that operate on the worklist; and that the class `Node` now has no knowledge of the worklist or the worklist algorithm. This strategy not only improves the modularity of the program, but also simplifies the data flow among different components. Due to this improvement, if the software developers decide to change the representation of the worklist, or to change the information that will be stored in the worklist, the modification will be confined within `prop()`. Thus, this maintenance task would be easier.

3. MODULARIZING COLLABORATING BEHAVIORS IN METHOD DESIGN

This section presents the core concepts for using aspects to modularize the collaborating behaviors in a method.

3.1 Aspectual Behavior Decomposition

Behavior advising can be viewed as event observation and handling [9, 18, 7, 6]: an aspect instance can observe the events that occur during the execution of other program components; the advices can be executed in response to the occurrences of these events. We propose a design strategy that uses behavior advising as a mechanism for coordinating the actions that must be performed for different tasks in a collaborating behavior. In our strategy, a suitable task will be identified and separated from the other tasks in the collaborating behavior. The actions for the identified task will be captured in an aspect and can be coordinated with actions performed for other tasks through behavior advising. Because such a coordination can be described completely within the description of the aspect, the other tasks in the collaborating behavior can be specified without any knowledge of the task captured by the aspect. This approach can achieve the separation of concerns in specifying the collaborating behavior.

Of course, a collaborating behavior can be decomposed with this strategy only if the identified task collaborates with other tasks through an observation/response protocol. In this case, we refer to the other tasks as the *driving* tasks and the identified task as a *reacting* task. More formally, a driving task progresses without direct control influence from the other tasks whereas a reacting task observes the progress of a driving task and perform it actions in response to the execution events. We refer to the strategy of identifying reacting tasks and capturing these tasks with aspects as the *aspectual behavior decomposition*.

3.2 Thread-Based Behavior Advising

We introduce a new execution model for behavior advising to better support the reasoning of programs that use aspects. In our model, the reacting task captured by an aspect and the cooresponding driving task are run in different threads; behavior advising is a mechanism for coordinating the actions of these two threads. We refer to the thread for the reacting task as an *aspectual* thread, and the thread for the driving task as the corresponding *driving* thread.

A driving thread defines the dynamic context in which an aspectual thread may observe the program execution. An aspectual thread may perform its actions in response to the actions performed only by its corresponding driving thread. The aspectual thread may also share data with the driving thread and indirectly influence the execution of the driving thread through these shared data. However, the as-

pectual thread can never substitute the actions for the driving thread or manipulate the arguments or return values for method invocations in the driving thread; the aspectual thread plays only an observer role. In addition, unlike existing techniques in which an exception raised by an aspect may interfere with the execution of other components, an exception raised by an aspectual thread will be propagated and handled without direct impact on the execution of the driving thread. This model will allow the driving thread to be reasoned about without referencing to the details of the aspectual thread.

An aspectual thread coordinates its actions with those performed by the corresponding driving thread through event observation. The aspectual thread must specify the types of execution events that it is interested and the response actions to be associated with such event types. During the program execution, when one such event occurs in the driving thread, the execution environment will suspend the driving thread and start executing an appropriate response action in the aspectual thread. After the response action of the aspectual thread terminates, the execution environment may resume the execution of the driving thread. This coordination mechanism ensures that only one of the two threads can be executed at a time. Therefore, the execution order for the actions of the two threads is deterministic.

An aspectual thread uses event queries to identify execution events that it is interested in observing. An event query characterizes the runtime circumstances under which a change of the program control or the program state will be considered as an event. The aspectual thread associates a response action to each event query. A pair of a event query and a response action is referred to as a *guarded command*. During the program execution, when an aspectual thread is being created, it will create a set of guarded commands. Each created guarded command will then submit its event query to the execution environment. When the execution environment detects an event identified by the event query of a guarded command, it will suspend the execution of the current thread and activate the aspectual thread to execute the response action associated with the event query.

Based on the purposes of the coordinations between an aspectual thread and a driving thread, we classify these coordinations into two categories: triggering and request. A triggering coordination allows the aspectual thread to perform an action immediately after a particular action has been performed in the driving thread. From the perspective of the driving thread, this coordination is optional; the task performed by the driving thread should be correct even if the aspectual thread does not perform any action. In this case, the driving thread does not need to have any mention of such a coordination. In contrast, a request coordination is used by the driving thread to solicit a guaranteed response from the aspectual thread. In this case, the driving thread must explicitly raise a *request* (a special event) by executing a special command. The driving thread will not be able to continue correctly unless the aspectual thread responds to the request according to a predefined protocol.

3.3 Statement-Level Aspectual Threads

We specify both the aspectual threads and the driving threads as statement-level threads. A statement-level thread is defined as a subcomponent in a thread statement. In a traditional language, a thread statement is typically delimited by a pair of special keywords (e.g., `cobegin`, `coend`). When the program control reaches the beginning of a thread statement, the execution of the current thread will be suspended and the child threads will be created from the thread definition subcomponents. These child threads will then start executing their actions. After all the child threads terminate, the execution of the thread statement terminates. The thread that contains the thread statement then resumes its execution and starts executing the next statement.

P1: <observation statement> :=
observe <bootstrap block> **during** <Java block>
P2: <bootstrap block> := {{<variable and command declaration list>}}
P3: <guarded command declaration> :=
on <event query expression> <Java block>
P4: <event query expression> := <event characterizing term> |
<event characterizing term> **&&** <boolean expression>
P5: <raise expression> := **raise** <request type id>(<expression list>)
P6: <request declaration> :=
request <return type> <request type id>(<type list>)
P7: <accept statement> := **accept** <expression>
P8: <decline statement> := **decline**

Figure 2: The syntax productions for defining the observation statement and the request coordination.

We extend Java programming language with an *observation* statement for defining the statement-level driving threads and aspectual threads. Figure 2 shows the syntax productions for defining such a statement. An observation statement (production $P1$) starts with a keyword **observe**, followed by a bootstrap block that defines the aspectual thread, another keyword **during**, and a regular Java block that defines the driving thread. A bootstrap block defines the guarded commands that specify the events that are interesting to the aspectual thread and the response actions that will be performed when an event is detected. A bootstrap block (production $P2$) consists of a list of declarations of local variables and guarded commands. These declarations are enclosed in a pair of double curly brackets to be distinguished from a regular Java block. The local variables declared in a bootstrap block are used to maintain the state information for the aspectual thread. They can be accessed by the guarded commands declared in this block.

A guarded command declaration (production $P3$) is specified with a keyword **on**, followed by an event query expression, a colon, and a regular Java block that defines the response action for the events being identified. An event query expression (production $P4$) is a boolean expression that contains an event characterizing term that may be conjuncted with a boolean subexpression as the filtering condition. The event characterizing terms for an initial set of primitive event types can be specified in the formats illustrated in Table 1 (the request will be explained in the next paragraph). Note that for some event types, the event characterizing term may be associated with a list of *observation interface variables* that allow the aspectual thread to accept values from the driving thread. The observation interface variables will be bound with values when an event described by the event characterizing term is detected. There is also an implicit observation interface variable `receiver` for accessing the receiver object for the execution context in which the event occurs, and `returnvalue` for accessing the return value of a

Event Type	*Format of Event Characterizing Term*	*Example*
method entry	`enter` ⟨class id⟩.⟨method id⟩(⟨obs-vars⟩†) `enter` ⟨variable⟩.⟨method id⟩(⟨obs-vars⟩)	`enter Node.addE(E e)` `enter p.addE(E e)`
method exit	`exit` ⟨class id⟩.⟨method signature⟩ `exit` ⟨variable⟩.⟨method signature⟩	`exit Node.addE(E)` `exit p.addE(E)`
reaching label	`reach` ⟨class id⟩.⟨method signature⟩.⟨label⟩ `reach` ⟨variable⟩.⟨method signature⟩.⟨label⟩	`reach Node.DFS().VISITING` `reach p.DFS().VISITING`
request	`request` ⟨type⟩ ⟨request type id⟩(⟨obs-vars⟩)	`request boolean Node.NotVisit(Node n)`

†⟨obs-vars⟩ are a list of observation interface variables that match the parameters of the associated method/request.

Table 1: Identifying primitive events.

method exit event.

We also extend Java programming language with a *raise* expression for raising a request that may be observed by an aspectual thread. A raise expression (production $P5$) starts with a keyword **raise**, followed by a request type id, and a list of expressions to pass values to the aspectual thread that will handle this request. A raise expression specifies a two-way communication channel between the driving thread and the aspectual thread. The list of expressions specified after the request type id in the raise expression will be evaluated when a request is raised; their values will be bound to the observation interface variables for the request. A raise expression can also return a value computed by the response action of the guarded command that handles this request. A request type id must be declared with a *request declaration* in a class. A request declaration (production $P6$) starts with a keyword **request**, followed by the request type id and a list of types for the values that can be passed to an aspectual thread when a request of this type is raised. The access to the request type id from outside of this class must be qualified with the name of this class.

The response action of the guarded command that handles a request may choose to accept or decline a request by executing an *accept* statement (production $P7$) or a *decline* statement (production $P8$) respectively. An accept statement may specify an expression whose value will be returned to the cooresponding raise expression.[2] Both accept statement and decline statement will terminate the response action. A request can be accepted or declined by at most one aspectual thread. If the request is declined, an unchecked exception of type `DeclinedRequestException` will be thrown at the place of the raise expression in the driving thread. If a request has not been accepted or declined by any aspectual thread, then an unchecked exception of type `UnhandledRequestException` will be thrown at the place of the corresponding raise expression in the driving thread. Both of these two types are subtypes of `RequestException`.

A driving thread or an aspectual thread created by an observation statement will be terminated in the following situations: (a) the thread executes a `break` statement defined in its description; (b) the thread executes a **return** statement defined in its description; or (c) the thread executes a statement that throws an uncaught exception. A driving thread will also be terminated when the control flow reaches the end of the Java block that defines this thread (situation (d)). In any of these situations, when one of the threads created for an observation statement is terminated, the other thread will also be terminated. In this case, their parent thread will resume the execution of the suspended observation statement. In the next step, the observation statement will terminate and the control will be transfered to the next action.

The next action to be performed after the termination of the observation statement is determined by the cause of this termination. For situations (a) or (d), the next action would be the statement that follows the observation statement. For situation (b), the next action would be the termination and the returning of the enclosing method. For situation (c), the next action would be the beginning of an appropriate exception handler.

3.4 Graph Searching: Another Example

This example illustrates the use of requests for coordinating actions. The program under construction needs to perform a depth-first-search on a graph to find the node whose key is the closest to a given integer value. Figure 3 shows an implementation of this searching using an observation statement. In the implementation, the graph traversal is captured in the driving thread specified in the observation statement, and the key checking is done by the aspectual thread. A request of `NotVisit` is raised at the beginning of `Node.DFS()` to determine whether a node should be visited.

In the aspectual thread, the first guarded command is used to ensure that each node will be visited only once; the second guarded command is used to check the key of a node against the designated value when the execution reaches the label "VISITING" in method `Node.DFS()`. Dudring the visit of a node, if the key matches the designated value, then the aspectual thread will terminate `Search()` with a `return` statement that returns the current node being visited. Otherwise, the aspectual thread will check to see whether the key of the current node being visited is closer to the value than that of the existing closest node. If that is the case, then the current node will become the new closest node. The traversal continues until a match is found or `Node.DFS()` terminates. In either case, the execution of the observation statement will be terminated. If `Node.DFS()` terminates normally, then the closest node will be return.

3.5 Extending Algorithm Frameworks

An observation statement can be viewed as a mechanism for extending the algorithm that is captured in the driving thread. From this perspective, a guarded command that observes and responds to requests can be viewed as a *required* extension whereas a guarded command that observes and responds to other execution events can be viewed as an *optional* extension. This mechanism allows software developers to define an algorithm framework within a method and then uses observation statements to extend this framework for different purposes. For example, `Node.DFS()` in Figure 3 can

[2]If the return type of the request is `void`, then the accept statement will not need to specify this expression.

```
class Node {
   int key;
   int getKey() throws NodeException {
     if( key<0 ) throw new NodeException();
     return key;
   }
   request boolean NotVisit(Node);
   void DFS() {
      if( raise NotVisit(this) ) return;
     VISITING:
      try{
        Node[] succs = getSuccs();
        for(int i=0; i<succs.length; i++)
          succs[i].DFS();
      } catch(Exception e) { ... }
   }
   ... //other declarations
}
...
Node Search(Node root, int value)
     throws NodeException {
  Node closest = root;
  observe {{
    Set visited = new HashSet();
    on request boolean Node.NotVisit(Node n): {
      if( visited.contains(n) )
         accept true;
      accept false;
    }
    on reach Node.DFS().VISITING: {
      visited.add(receiver);
      if( receiver.getKey()== value )
         return receiver;
      else
         if( closest.getKey()-value
            >receiver.getKey()-value )
            closet = receiver;
    }
  }} during {
     root.DFS();
  }
  return closest;
}
```

Figure 3: Implementing a graph searching algorithm with an observation statement.

be viewed as a depth-first-search algorithm framework, and the observation statement in `Search()` extends this framework for searching the graph; `Node.DFS()` can certainly be extended for purposes such as searching based on a different criterion or for searching a tree instead of a graph.

An algorithm framework can be extended with observation statements in a hierarchical way that resembles the way in which a class gets extended. Figure 4 shows `SearchInRange()` that extends `Search()` to control the searching not to go beyond nodes whose keys are greater than 100. In this case, a request raised by `Node.DFS()` will first be processed by the guarded command in `SearchInRange()`. This guarded command first checks whether the key of the node is greater than 100. If this is the case, the guarded command will accept the request with true. Because only one aspectual thread can accept a request, the guarded command in `Search()` for the request will be skipped and the program control will be transfered back to `Node.DFS()`. However, if the key of the current node is not greater than 100, then the request will also be processed by the first guarded command

```
Node SearchInRange(Node root, int value)
                    throws NodeException {
  observe {{
    on request boolean Node.NotVisit(Node n): {
      if( n.getKey()>100 )
          accept true;
    }
  }} during {
     return Search(root,value);
  }
}
```

Figure 4: Extending `Search()`.

in `Search()`. This example shows that a guarded command of an aspectual thread for handling requests may conditionally "override" a guarded command of another aspectual thread for handling the same type of request.

4. COORDINATING NESTED THREADS

A driving thread or a response action of an aspectual thread in an observation statement are specified with a regular Java block. Such a block may contain another observation statement or a method call that can lead to the execution of another observation statement (e.g., the call to `Search()` in the driving thread in `SearchInRange()` of Figure 4). That is, when a driving (or aspectual) thread is being executed, it may execute an observation statement and create a new pair of child threads. We refer to these created threads as *nested* threads.

The presence of nested threads complicates the coordination among the actions of different threads. An observation statement executed by a thread is used to implement some of the responsibilities of this thread. Therefore, the two child threads created by the execution of this statement should be considered as integral parts of the containing thread. That is, any action performed by the child threads should be considered as an action performed by the parent thread. For this reason, we have the following rule:

> **Observation rule.** An aspectual thread can and only can observe events that occur in its corresponding driving thread or any descendant thread of the driving thread.

Because of this rule, the events in a nested thread may be observed by more than one aspectual thread. In this case, the response actions in these aspectual threads must be performed in a predefined order so that the combined effect of these actions can be reasoned about consistently.

To explain how the actions in different threads should be coordinated in the presence of nested threads, we use a tree to represent the creation relationship among the currently existing threads. On the tree, the root represent the main thread of the program, and other nodes represent the driving threads or the aspectual threads created by the observation statements; and the edges represent the creation relationship among these threads.[3] At a particular point of time, a thread can have children only if it is suspended on an observation statement. Because a thread is executed sequentially, only one statement in this thread can be suspended at this

[3]If the program creates regular Java threads other than the main thread, then we will have a set of such trees, each of which is rooted with a regular Java thread.

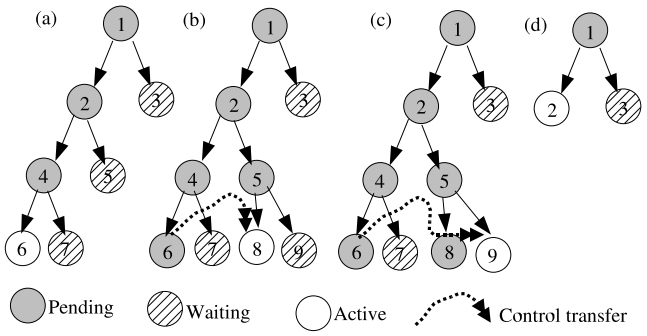

Figure 5: The snapshots of the activation trees at various points of time.

point of time. Therefore, a node on this tree either have no child, or have exactly two children that represent the driving thread and the aspectual thread created by the suspended observation statement. With this property, we can arrange the tree so that the left child of a node represents the driving thread and the right child of this node represents the aspectual thread. We refer to this tree as the *activation tree* for the threads.

Given the arrangement of the nodes in the activation tree, if a node represents an aspectual thread, then it should always have a left sibling node that represents the corresponding driving thread. According to the observation rule, we have the following:

> **Corollary 1.** Given an activation tree at a point of time during execution, an aspectual thread represented by a node n in the tree can and only can observe the events in threads represented the nodes in the subtree rooted at n's left sibling.

We can infer that, if an event occurs in the thread represented by a node n on the activation tree, then only the aspectual threads represented by the right siblings of n and n's ancestors can observe this event. We refer to these aspectual threads as the *eligible* aspectual threads for the event. Because each node can have at most one right sibling, for a particular event, there is at most one eligible aspectual thread at each level of the activation tree. Therefore, the eligible aspectual threads for this event can be ordered based on their distances from the root of the tree.

In our runtime model, when more than one eligible aspectual thread is interested in an event, then the event queries and the response actions of these threads will be executed in the order based on their distances to the root. Because the response action for an aspectual thread may terminate the execution of the corresponding observation statement, this arrangement will give the priority to a higher level aspectual thread to make such a decision. Note that, for a request, once an eligible aspectual thread accepts or declines this request, the remaining eligible aspectual threads will not allow to execute their response actions. This ensures that at most one aspectual thread can accept or decline a request.

Figure 5 uses activation trees to illustrate a scenario for coordinating the actions in different threads in the presence of nested threads. Figure 5(a) shows the activation tree after an event occurs in the currently active thread that is represented by node 6 (from now on, we use the node numbers to identify the threads). For this event, the aspectual threads 3, 5, 7 are eligible aspectual threads. Suppose that thread 5 is interested in this event. Then, thread 6 will be suspended and thread 5 will evaluate its event queries and execute one of its response blocks. Suppose that the response block executes an observation statement. In this case, thread 5 will create a pair of driving child thread and aspectual child thread. Figure 5(b) shows the activation tree after this pair of threads (numbered 8 and 9) have been created; at this point of time, thread 5 is suspended and the control is transfered to thread 8.

During the execution of thread 8, an event occurs. In this case, only threads 3 and 9 are eligible for this event. Suppose that thread 9 is interested in the event. Thread 8 is suspended and thread 9 will execute a response action. The response action throws an exception of type `E`. Figure 5(c) shows the status of the threads when this occurs. Suppose that, of all the threads, threads 2, 4, 8 can handle exceptions of type `E`. Because an action performed by a thread is considered as part of its ancestor threads, this exception should be handled by thread 2. Therefore, all the threads represented by the descendants of the node 2 in the tree should be terminated and the control should be brought back to thread 2. Figure 5 (d) shows the activation tree after the control is back to thread 2.

5. A SINGLE-STACK IMPLEMENTATION OF OBSERVATION STATEMENTS

A program that uses observation statements semantically is a restricted multi-threaded system. In general, because different threads in a multi-threaded system can progress concurrently and even nondeterministically, each thread must use a separate stack to maintain the activation records for its unreturned method invocations. Maintaining these stacks and switching the execution contexts from one thread to another may incur significant time and space overhead.

In our runtime model, only one thread can be executing at a particular point of time. Any other thread is either suspended or in a state in which it is waiting for the occurrence of events. An executing thread may give up the execution control in two situations. First, the current thread starts executing an observation statement. In this case, the current thread will create two child threads, suspend itself, and transfer the control to the driving thread. The current thread remains suspended until the two child threads terminate. Second, the current thread has just performed an action that is interesting to an aspectual thread. In this case, the current thread will suspend itself and transfer the control to the aspectual thread. The current thread remains suspended until the aspectual thread finishes executing the response action and enters the waiting state. In both cases, after a thread gives up the execution control to another thread, the first thread can reclaim the execution control only if the second thread is terminated or has entered the waiting state. Thus, all method invocations of the second thread should have been returned. This characteristic allows different threads to be run on the same stack.

In our implementation, we translate the descriptions of the threads in an observation statement into regular methods and use method calls, method returns, and exception handling to transfer the control among the threads. This implementation approach allows the program with observation statements to be run on a standard Java Virtual Ma-

```
class ReturnException
              extends Exception {
    Object objReturn;
    ReturnException(Object obj) {
       objReturn = obj;
    }
}
```

```
class CarrierException extends Error  {
   Exception objExcept;
   ThreadEnvironment tENV;
   CarrierException(Exception e, ThreadEnvironment env) {
      objExcept = e;  tENV = env;
   }
}
```

(a) special exceptions

```
1 class Observe1 extends ThreadEnvironment {
2    Node closest, root;
3    int value;
4    Set visited = new HashSet();
5    Observe1() {
6      // subscribe the event queries declared
7      // in the guarded commands
8    }
9    void destroy() { /*unsubscribe event queries*/ }
10   void driving() {
11     root.DFS();
12   }
13   void grdCmd1(NotVisitRequest req) {
14     if( visited.contains(req.n) ) {
15       req.accept(true); return;
16     }
17     req.accept(false); return;
18   }
19   void grdCmd2(LabelEvent le) {
20     try {
21       if( le.receiver.getKey()==value )
22         throw new ReturnException(le.receiver);
23       else if( closest.getKey()-value
24               > le.receiver.getKey()-value )
25       closest = le.receiver;
26     } catch(Exception e) {
27       throw new CarrierException(e,this);
28     }
29   }
30 } // end of class Observe1
```

```
31 void Search(Node root,int value)
                      throws NodeException {
32  Node closest = root;
33  Observe1 ob1;
34  try{
35    // initializing the environment
36    ob1 =  new Observe1();
37    ob1.closest = closest;
38    ob1.root = root; ob1.value = value;
39    //transfer control to driving method
40    ob1.driving();
41  }
42  catch(CarrierException carE) {
43    if( carE.tENV != ob1 )// not for
44       throw carE;       // this thread
45    Exception e = carE.objExcept;
46    if( e instanceof ReturnException )
47        return (Node)
48          ((ReturnException)e).objReturn;
49    else // other types of exceptions
50      throw e;
51  }
52  finally {
53    closest = ob1.closest;
54    ob1.destroy();
55  }
56  return closest;
57 } // end of Search
```

(b) The code for Search() **after translation**

Figure 6: Implementing an observation statement with standard Java constructs.

chine. The approach can also reduce the time and space overhead required for coordinating the threads created by the observation statements.

Like other AOP techniques, the behavior advising in our technique requires the support from an event observation and dispatching (EOD) environment. A raise expression will be translated into a call to a special method to notify the EOD environment about the request. Probing statements must be inserted at appropriate locations in the program to detect the execution events that may be interesting to some aspects. These probe statements will notify the EOD environment. When an EOD environment is notified with an event, it will look up the eligible aspectual threads for this event and invoke appropriate methods for these threads to process the event.

Example. Figure 6(b) sketches the Java code for implementing the observation statement in `Search()` of Figure 3. In Figure 6(b), class `Observe1` is introduced to capture the threads described in the observation statement. `Observe1` extends `ThreadEnvironment`. It contains method `driving()` that captures the driving thread and contains methods `grdCmd1()` and `grdCmd2()` that capture the response actions for the two guarded commands declared in the bootstrap block in the observation statement. Both `grdCmd1()` and `grdCmd2()` take an instance of the corresponding event as input. The observation interface variables for the event are represented as fields for the input instance.

The response block for the second guarded command in Figure 3 may execute a `return` statement to terminate `Search()`. This effect is emulated with the throwing of an exception of `ReturnException` in `grdCmd2()` (line 22 in Figure 6(b)). However, because `grdCmd2()` will be invoked on the stack that is also used by `driving()` before `driving()` returns, this exception or any other exception thrown in `grdCmd2()` will be intercepted by the `catch()` clause in `Node.DFS()` shown in Figure 3. We wrap an exception thrown by `grdCmd2()` with an instance of `CarrierException` to avoid any possible interception. `CarrierException` (Figure 6(a)) extends `Error` to become an unchecked exception.[4] An instance of `CarrierException` also contains a reference to the instance of `Observe1` so that the exception will be delivered to the right thread environment in the presence of nested threads.

`Observe1` also contains fields for maintaining the data required for executing the statements in the three methods. It contains a field `visited` for the local variable declared in the bootstrap block of the observation statement in Figure 3. It also contains *shadow* fields for the three local vari-

[4]We assume that `Error` will not be caught by `catch` clauses in the original program. A better solution would be to let `CarrierException` extend `Throwable`. However, the Java compiler needs to be extended to recognize it as an unchecked exception in this case.

ables `value`, `closest` and `root` whose values may be used or modified by the driving thread or the aspectual thread.

The observation statement in `Search()` of Figure 3 is translated into the `try` structure at lines 34-55 in Figure 6(b). This `try` structure is responsible for instantiating `Observe1`, initializing the shadow fields, invoking `driving()`, and assisting the EOD environment to bring the control-flow from `grdCmd2()` back to `Search()`. The EOD environment is responsible for invoking `grdCmd1()` and `grdCmd2()` when events are detected during the execution of `driving()`. If an exception of `CarrierException` is thrown before `driving()` returns, this exception can be caught at line 42 and processed by the `catch` block. The `catch` block first verifies that this exception is indeed thrown by a method of the instance for `Observe1`. In this case, the carried exception is extracted. If the extracted exception is an instance of `ReturnException`, then a return statement will be executed. Otherwise, the carried exception will be re-thrown. A `finally` block is also included in the `try` structure to copy the updated value of the shadow field to the local variable `closest`, and to invoke `destroy()` on the instance of `Observe1` to unsubscribe the event queries for the guarded commands.

Prototype and evaluation. We have developed a prototype translator for translating the observation statements into standard Java constructs. The translator is built using MetaBorg[5]. The probing statements for detecting execution events are inserted into the classes using the compiler for AspectJ. Using the prototype, we successfully compiled several programs that contain observation statements into Java class files.

We have also developed a prototype event observation and dispatching environment to support the behavior advising. Event observation and dispatching may incur overhead. Compring the performance of the example program in Figure 3 to a hand-coded version of the program revealed that, to search a randomly-generated tree of $2^{16} - 1$ nodes, using the observation statement may introduce averagely 95% additional cost to `Search()`. Note that, in `Search()`, only a few simple statements will be executed when a node is being visited. When more expensive computation is required during the visiting of a node, the overhead is expected to be much lower in terms of percentage. Optimization techniques may also be developed to reduce such overhead.

We have also performed a limited case study to evaluate the overhead that may incur in a real program. In the case study, we have modified the Java analysis framework mentioned in Section 2 to introduce four observation statements for improving the program structure. These observation statements are on the frequent path. Our study shows that using these statements introduce about 2% overhead to the total time required for the program. More studies are required to thoroughly evaluate such an overhead.

6. RELATED WORK

Event observation has been used at the architecture level as a mean for integrating the behaviors specified by different components in many systems. This design paradigm has been referred to as *event-based programming*, or as *implicit invocation* in contrast to the *explicit invocation* through regular procedure calls. Constructs for event declaration, event announcement, and the binding between the events and the handling procedures have been proposed in some languages to support general purpose implicit invocations [15, 11].

Behavior advising supported by the aspect constructs in existing AOP languages can be viewed as an extended form of event observation and handling or implicit invocation (e.g., [9, 18, 19, 7, 6, 21]). These aspect constructs and the language constructs proposed in [15, 11] make different trade-offs in supporting event observation and handling. By relying on the program structures and program states for identifying events, aspects can observe events that do not require explicit announcements. This brings a key advantage (the *obliviousness* [8]) of AOP: the behaviors under observation can be specified without any mention of the coordination with the aspects that handle the events. Of course, this creates a dependency between an aspect and the program structures of other components: a change to the program structure may break aspects. In contrast, language constructs that rely on explicitly announced events will not have such a problem. AOP techniques also extend the traditional implicit invocation to allow advices in an aspect to manipulate the inputs or the return value for a method invocation or even substitute the method invocation with another action. This capability increases both the expression power and complexity of behavior advising.

The statement-level aspects presented in this paper are designed with yet a different set of trade-offs to better support the use of event observation and handling for implementing the collaborating behaviors in method design. They differ from the existing aspect constructs in several ways. Unlike the existing AOP techniques in which an aspect must be specified as a separate module, a statement-level aspect is specified as a subcomponent for an observation statement, and thus, becomes an integral part of a method. In addition, the thread-based runtime model for aspects prevents an aspect from directly affecting the control-flow in the implementation of the driving task. Furthermore, requests must be explicitly raised in the driving task if the correctness of the driving task depends on a specific response from the the aspect. These features allow the implementation of the driving task to be reasoned about without referring to the details of the aspect. Of course, with these differences, statement-level aspects may be less effective than the existing aspects in solving some design problems other than the implementation of the collaborating behaviors. For example, unlike AspectJ, our technique will not allow an aspect to be added into a program without modifying the program. Therefore, we view that the statement-level aspects compliment other forms of aspects. Using different forms of aspects together may allow software developers to form effective strategies in implementing the program.

Clifton and Leavens [4, 3] distinguish two different kinds of aspects: observers(spectators) and assistants. An observer cannot change the behaviors defined in other modules whereas an assistant can change the behaviors defined in other modules. To allow "understanding a system one module at a time", they proposed that an assistant must be "accepted" into a module in order to change the behavior defined in this module. In contrast, the use of an observer has no restriction. Our statement-level aspects can be viewed as a less restricted observer: a statement-level aspect cannot directly impact the control-flow of the behaviors being observed; however, the statement-level aspect can

[5]http://www.metaborg.org/.

indirectly influence the execution of the behaviors being observed through shared data or requests. Such a restriction is automatically enforced by our thread-based runtime model for aspect.

Aldrich [1] proposed open modules that allow an external aspect to advise the behaviors of a module only through observation of the interactions between the module and the outside world or through explicitly exported pointcuts. This approach can prevent an aspect from directly depending on the implementation details of other modules. Therefore, changing the internal implementation of a module will not break the aspects. This approach can benefit any form of aspect, including our statement-level aspects.

Design patterns such as the *visitor* pattern and the *template method* pattern [10] may be used to decompose the tasks in a collaborating behavior to achieve a certain degree of separation of concerns. These patterns rely on the collaborations of objects. However, using method calls to coordinate the actions among these objects can lead to code scattering and tangling. It is interesting to see how observation statements can be used to eliminate this code scattering and tangling in the implementation of these patterns.

7. CONCLUSION

This paper presents a statement-level aspect construct that is designed to support the implementation of collaborating behavior in class methods. This new construct is designed with a more balanced view between the expressiveness and the comprehensibility. Like an aspect proposed by existing AOP techniques, a statement-level aspect can coordinate its actions with other program components through event observation. With examples, we show that this capability can help to prevent the scattering of the coordination statements for implementing collaborating behaviors in class methods. However, unlike an aspect proposed in existing techniques that may directly affect the control-flow of other program components, a statement-level aspect can only affect the execution of other program components through shared data. This restriction may ease the reasoning of the dynamic behaviors of a program.

Much remains to be done. In the future, we will continue improving the quality and the usability of our prototype. Using the prototype, we will perform studies to evaluate, in real software development projects, the usefulness and the usability of the proposed program constructs. We will further study the interaction patterns in the collaborating behaviors that present in programs and evolve the proposed program constructs to deal with more complicated patterns.

Acknowledgements. We thank Gary Leavens for his insightful comments on this work and the anonymous reviewers for the comments on an early version of this paper.

8. REFERENCES

[1] J. Aldrich. Open modules: Modular reasoning about advice. Technical Report CMU-ISRI-04-141, CMU, 2004.

[2] AspectJ. http://www.eclipse.org/aspectj/.

[3] C. Clifton and G. T. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *Foundations of Aspect Languages*, 2002.

[4] C. Clifton and G. T. Leavens. Obliviousness, modular reasoning, and the behavioral subtyping analogy. Technical Report TR 03-01A, Iowa State University, 2003.

[5] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using aspectc to improve the modularity of path-specific customization in operating system code. In *ESEC/FSE-9*, 2001.

[6] R. Douence, P. Fradet, and M. Sudholt. Composition, reuse and interaction analysis of stateful aspects. In *Proceedings of the 3rd international conference on Aspect-oriented software development*, pages 141–150. ACM Press, 2004.

[7] R. Douence and M. Sudholt. A model and a tool for event-based aspect-oriented programming (eaop). Technical Report 02/11/INFO, Ecole des Mines de Nantes, 2002.

[8] R. Filman and D. Friedman. Aspect-oriented programming is quantification and obliviousness. In *Workshop on Advanced Separation of Concerns, OOPSLA 2000*, 2000.

[9] R. E. Filman and K. Haveltund. Realizing aspects by transforming for events. In *Automated Software Engineering (ASE)*, Sept. 2002.

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

[11] D. Garlan and C. Scott. Adding implicit invocation to traditional programming languages. In *Proceedings of the 15th international conference on Software Engineering*, pages 447–455, 1993.

[12] M. Kersten and G. C. Murphy. Atlas: A case study in building a web-based learning environment using aspect-oriented programming. In *Proc. ACM Conf. Object-oriented Programming, Systems, Languages, and Applications*, pages 340–352, 1999.

[13] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *The European Conference on Object-Oriented Programming*, 2001.

[14] D. Liang and Y. Guo. Effective language support for aspectual behavior decomposition. Technical report, University of Minnesota, 2005.

[15] D. Notkin, D. Garlan, W. G. Griswold, and K. Sullivan. Adding implicit invocation to languages: Three aproaches. In *Proc. JSSST Symp. Object Technologies for Advanced Software*, Nov 1993.

[16] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *FSE/ESEC*, pages 297–306, 2003.

[17] H. Rajan and K. Sullivan. Classpects: Unifying aspect- and object-oriented language design. Technical report, University of Virginia, 2004.

[18] R. J. Walker and G. C. Murphy. Joinpoints as ordered events: Towards applying implicit context to aspect-orientation. In *Proceedings of the 23rd International Conference on Software Engineering*, May 2001.

[19] R. J. Walker and K. Viggers. Implementing protocols via declarative event patterns. In *12th International Symposium on the Foundation of Software Engineering*, 2004.

[20] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[21] J. Xu, H. Rajan, and K. Sullivan. Understanding aspects via implicit invocation. In *Automated Software Engineering*, 2004.