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 05-011

Effective Language Support for Aspectual Behavior Decomposition

Donglin Liang and Yi Guo

April 04, 2005

Effective Language Support for Aspectual Behavior Decomposition

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

Abstract

Aspects have been proposed as a new program abstraction to achieve better separation of concerns during software design. Although many aspect-oriented programing (AOP) techniques have been proposed to support the use of aspects, the conceptual framework for characterizing the design problems that can be solved using aspects is not well formulated. The lack of a well-formulated framework makes it difficult for software developers to identify the places where aspects can be best used; it also makes it difficult to evaluate and to improve the techniques that support aspects. In this paper, we propose to formulate a conceptual framework with the concept of collaborating behaviors. Based on this perspective, we discuss the concept of aspectual behavior decomposition and the limitations with the existing AOP techniques in supporting such a decomposition strategy. We also give a brief overview of a new AOP technique that illustrates how some of these limitations can be overcome.

1 Introduction

Software systems that we develop have increasingly complex behaviors. To design such a system, software developers typically employ a series of decomposition strategies to divide the behaviors and the data involved into smaller units so that each unit can be described by a specific programming abstraction (e.g., a procedure, an object). With appropriate integration mechanisms, the behavior and/or data units specified by these abstractions can be put together during runtime to provide the required functionalities. Ideally, the decomposition should achieve the *separation of concerns*: in the resulted program, the parts of software relevant to each important concern (e.g., a concept, a goal, or a purpose) should be separated and captured with appropriate programming abstractions [15]. This arrangement would allow the codes for these concerns to be developed, verified, and maintained separately. Achieving this separation is vital for building and maintaining high quality software systems.

Program decomposition strategies must be formed around specific programming constructs. In the last few years, much research has been carried out to search for new programming constructs to support more effective program decomposition strategies. Such an effort has resulted in several innovative programming paradigms (e.g., [2, 11, 14, 19]), each of which is developed to address the specific limitations with the traditional programming techniques in the modularization of some particular concerns. Among these new paradigms, Aspect-Oriented Programming (AOP) [11], made popular by AspectJ [1], has been shown quite promising in modularizing various concerns in realistic systems (e.g., [3, 9]).[1]

AOP provides aspects as a new kind of program constructs for modularizing program specification. The most basic form of aspect is a program structure that can be imposed on classes declared in a *base program* for extending their definitions. An aspect has two distinctive features. First, it can contain *static crosscutting* declarations that specify various modifications (e.g., adding members, adding interfaces being implemented) to the structure of existing classes. Second, it can also contain advice declarations that specify code segments (referred to as *advices*) that will be attached to various syntactic locations (referred to as *joinpoints*) in the methods of the existing classes to modify (or *advise*) the behaviors captured by these methods. In this paper, we will focus our discussion on the behavior advising. Many recent developments in AOP (e.g., [4, 17, 18]) are also focusing on the behavior advising.

The basic form of aspect can be viewed as *static*, in the sense that the introduction and the application of a aspect cannot be controlled by the program execution. This concept was later extended to be *dynamic*. In the dynamic setting, an aspect must be instantiated for the advising to take effect. The created aspect instance may then modify the behaviors of one or more class instances (*instance-level advis-*

---

[1] Sometimes, AOP may also be used to refer to paradigms such as [2, 14]. In this paper, we use it to refer to AspectJ-like AOP.

ing [17]). AspectJ 1.2 [1] only allows aspect instances to be created at specific joinpoints. EOS [17], on the other hand, allows program statements to create aspect instances from aspect declarations in a similar way as the creation of class instances; it also allows the aspect instances to be passed around through reference variables. EOS-U [18] goes one step further to unify aspects and regular classes into a single concept—classpect. A method declared in a classpect can be invoked through method calls or can be used to advise the actions performed in the program. This unification is claimed to improve conceptual integrity.

Aspects were originally proposed to modularize the implementation of *crosscutting concerns*—"concerns that inherently crosscut the natural modularity of the rest of implementation" [10]. However, this vision may need to be expanded to accommodate the design problems that may be solved by the recent and future development of AOP. As in EOS and EOS-U, aspects now become a general abstraction like objects that can be used to form effective decomposition strategies during the design of any part of the system. The trend indicates that aspects should not be considered as being outside of the scope of "natural modularity" and are used only for non-core functionalities. Instead, they can be used to achieve the "natural modularity" in the implementation of both core or non-core functionalities. Of course, further development may be required for AOP technology to support this vision.

This paper presents our research in searching for new AOP techniques in supporting such a vision. We revisit the conceptual framework that characterizes the design problems that can be solved by using the advising capabilities of aspects. As the result, we propose to formulate such a framework with the concept of collaborating behaviors and to view aspects as program constructs for supporting effective decomposition strategies for capturing such behaviors. Based on this perspective, we formalize the concept of aspectual behavior decomposition and discuss the limitations with the existing AOP techniques in supporting such a decomposition strategy. We finally give a brief overview of a new AOP technique that illustrates how some of these limitations can be overcome.

2 Why Aspects? A Behavior Perspective

To enable the effective use of a particular type of program abstraction during software design, there must be a conceptual framework for characterizing the design problems that can be solved by using this type of program abstraction. The framework should help the software developers to identify, at various stages of design, the places where the program abstraction can be best used. The framework can also guide researchers to evolve the program abstraction and to develop programming constructs and methodologies that can best fit the programming needs. This section will discuss such a conceptual framework for aspects.

Problems with the concept of crosscutting concerns. In current AOP literature, the design problems that can be solved by using aspects are characterized with the concept of crosscutting concerns. A careful examination with this concept reveals several problems. First, in current AOP literature, the crosscutting concerns are not well defined. Crosscutting concerns were earlier referred to as "aspects" that can "cross-cut system's basic functionality" [11]. Later, in the overview of AspectJ, Kiczales et. al. [10] define crosscutting concerns to be "concerns that inherently crosscut the natural modularity of the rest of the implementation". Laddad, in his book "AspectJ in action" [12], defines crosscutting concerns to be "system-wide concerns that span multiple modules". Many other publications define this concept in a similar way. Unfortunately, these definitions contain concepts that are not well-formulated. For this reason, they may be difficult to use during design to identify the situations in which aspects would be useful. What is the system's basic functionality? What is a concern? What is crosscutting? What is "the natural modularity of the implementation"? Second, these definitions of crosscutting concerns do not seem to keep up with the evolution of AOP techniques: they may be too restricted for describing the problems that may be solved by existing and future AOP proposals. For example, what if a concern that causes code scattering and tangling is a part of the system's basic functionality? Or shouldn't the modularization with aspects consider natural? Third, the term "crosscutting concern" itself seems inconsistent. In current literature, a concern usually refers to a concept, a goal, or a purpose, etc [15]. A concern is supposed to be about what, not about how. In contrast, crosscutting is about the structural relationship of several things in the implementation. Without knowing how the concern can be implemented, it may not be easy to see why a particular concern may crosscut the modules.

In summary, although the crosscutting concern may be a good concept for communicating the essence of AOP to general public, it may not be a good concept for formulating a conceptual framework to guide the usuage of AOP constructs and the evolution of AOP. We must find another concept for characterizing the design problems that can be solved by AOP. To allow the software developers to identify such problems at various design stages, the definition of this concept should only refer to entities/concepts that are common at these stages.[2] Ideally, this concept should be grounded on software design terminologies that have been well understood by software developers.

---

[2] Many definitions of the crosscutting concern do not meet this requirement because they refer to modules or modularity that are only available at the later stages of design.

Collaborating behaviors. We propose the *collaborating behaviors* as an alternative concept for formulating an important set of design problems that can be solved using the advising capability of aspects. A collaborating behavior refers to the situation in which a system must carry out several tasks within a certain period of time during execution. For each task, the system must perform a sequence of actions to achieve a particular goal. In many cases, these tasks must collaborate according to some specific protocol: the actions required for different tasks must share data properly; and these actions must be coordinated to ensure that they are performed in certain order. For example, to provide the logging capability for a system, a logging task must collaborate with the primary functional tasks of the system according to the following protocol: whenever an action is performed for a primary functional task, the logging task must perform an action that outputs information to record the progress of the primary functional task. To achieve the separation of concerns in describing a collaborating behavior, different tasks should be described with separated program abstractions; and the description of the collaborating protocol for these tasks should also be localized.

We argue that the existence of collaborating behaviors is an important cause for the code scattering and tangling problems that are common in systems implemented with traditional programming languages.[3] Avoiding such scattering and tangling was the major motivation for the introduction of AOP. In a programming language, actions will be specified with statements. These statements must be composed appropriately so that the specified actions can be performed in the right order during program execution. In many existing programming languages, statements can be composed using the sequential operator or composite statements (e.g., `if` statement). A sequence of statements can be encapsulated within a method (procedure) so that these statements, as a whole, can be called into being during program execution. To implement a collaborating behavior with such a language, the actions required for each task often get divided into smaller chunks to fit within individual methods so that these actions can be properly scheduled with method calls. To facilitate the effective control transfer and data sharing among the actions for different tasks, these actions may be collocated within the same method.

The above implementation scheme has drawbacks. First, the codes that describe the actions for a given task may be scattered in several methods. Often, there is no well-defined boundaries to separate the descriptions of the actions that belong to different tasks. In addition, to provide the necessary data flow across method boundaries for the actions of a particular task, additional parameters may need to be introduced in methods that are not directly related to the implementation this task. Thus, it is difficult to maintain and to reason about the codes that implement a particular task. Second, the mechanisms (control flow or method calls) that coordinate the execution of the actions among different tasks are also scattered in different methods. These mechanisms are often tangled with similar program constructs that are used for other purposes. Therefore, it may be quite challenging to reason about the interactions among the collaborating tasks.

```
class Node {
   Set ents;
   void addE(E e,WList wlist)
            throws WListExcpt {
     if( !isIn(e) ) {
        ents.add(e);
        if(wlist!=null)
          wlist.add(this);
          // exception of WListExcpt
          // may be thrown by add()
     }
   }
   void process(WList wlist)
             throws WListExcpt {
     // generate new entities,
     // propagate these entities
     // by invoking addE()
     // on the children
   }
   boolean isIn(E e) {
     return ents.contains(e);
   }
   // other methods and fields
}
void prop(Node root, E ee) {
   try {
     root.addE(ee,null);
     WList wlist = new WList(root);
     while( !wlist.empty() ) {
        Node n = wlist.getNext();
        n.process(wlist);
     }
   } catch(WListExcpt e) { ... }
}
```

Figure 1. The implementation of a worklist algorithm.

Collaborating behaviors are common in many software systems. *Besides well-studied crosscutting concerns, such as logging or synchronization, there may be many other reasons to account for this kind of behavior.* Figure 1 shows an excerpt of a Java program that contains a collaborating behavior that may not be considered as being caused by crosscutting concerns. The program under construction contains an algorithm that propagates entities of type `E` from the root of a tree to all the nodes in the tree. This algorithm is a skeleton of an actual algorithm implemented in a Java alias analysis framework. In the program, the nodes of the tree are instances of class `Node` and the propagation algorithm is captured in `prop()`. In general,

---

[3] There may be other causes. The discussion of a full spectrum of these causes is outside of the scope of this paper.

when a new entity is added to a node, additional entities may be created and propagated to the children of this node (described in `Node.process()`). Adding new entities to the children, however, would require these children to be processed in a similar way. To implement this behavior, 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 in their signatures. Also note that `Node.process()` and `Node.addE()` may be invoked by methods that are not shown here.

In this example, the worklist maintaining and the node processing can be viewed as two collaborating tasks whose actions must be coordinated appropriately to ensure the correctness of the implementation. The program excerpt in Figure 1 shows that the statements required for maintaining the worklist are now scattered in `prop()` and `Node.addE()` to ensure that the worklist is updated appropriately. To allow the worklist to be updated in `Node.addE()`, additional parameters must be introduced in various methods to pass a reference of the worklist down the call chain from `prop()` to `Node.addE()`. To further complicate the matter, 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 has to 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. 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`.

Summary. In searching for new programming paradigms to support more effective program decomposition strategies, we propose to focus our attentions on the collaborating behaviors, instead of the various concerns that may lead to such behaviors. Thus, *we view aspects as a new type of program abstraction developed for better modularizing collaborating behaviors*. This perspective may lead to the development of more general aspect constructs for modulizing the implementation related a broader class of concerns than the “cross-cutting” ones. In addition, behaviors are identified and analyzed at all levels of design. Therefore, characterizing the usage of aspects with behaviors would allow software developers to systematically identify the situations for using aspects throughout the design stages. This is an important characteristics of a general software development paradigm, such as the object-oriented paradigm.

3 Aspectual Decomposition of Collaborating Behaviors

In this section, we will discuss the behavior decomposition strategies enabled by aspects. An aspect can be viewed as a program entity that captures a set of actions (specified by the advice codes) that can be coordinated, through an advising mechanism, with other actions performed by the program.[4] Based on the join-point-advice bindings specified in the aspect, the advising mechanism can invoke appropriate advice codes in the aspect when the program control reaches various points of the program, without explicitly calling these advice codes at the particular program points. This mechanism is very similar to the *implicit invocation* that has been formulated and studied (e.g., [8]) in the event-based programming paradigm. Indeed, advising can be viewed as event observation and handling [7, 20, 5, 4]. The various points of time during a program execution can be identified with different types of execution events. The invocation of a piece of advice code can be viewed as the handling of a particular type of execution event.

We propose the following decomposing strategy for a collaborating behavior. We identify a specific task within a collaborating behavior and use an aspect for capturing the actions that will be performed for this task. These actions can be coordinated with actions performed for other tasks through the aspect advising mechanism. Because such a coordination is specified with the join-point-advice bindings that are completely captured in the aspect, the other tasks in the collaborating behavior can be specified without any knowledge of the task captured by the aspect. Therefore, using this decomposition strategy, the specification of this task and the collaborating protocol can be completely separated from the descriptions of the other tasks. This approach can achieve the separation of concerns.

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. We refer to a collaborating behavior that contains this type of task as the *reactive collaborating behavior*. More formally, a reactive collaborating behavior contains the following two types of tasks. The first type of tasks progress without direct control influence from the others; whereas the second type of tasks would observe the progress of the first kind of tasks and perform their actions in response to the execution events. We refer to the first kind of tasks as the *driving* tasks and the second kind of tasks as the *reacting* tasks. We refer to the process of identifying reacting tasks and capturing these tasks with aspects as the *aspectual behavior decomposition*.

Aspectual behavior decomposition strategies should be

---

[4] As mentioned in Introduction, we only focus on the advising capabilities of aspects in this paper.

allowed to apply at different design levels (e.g., system design, object design, or even method design). These strategies must be supported by appropriate aspect integration constructs that specify how the identified aspects can be integrated with other program entities during runtime to provide the required functionalities. Some AOP techniques (e.g., [1, 4]) only allow aspects to be integrated with other programming entities at the system level. In these techniques, aspects are defined as entities that can be integrated with the base program as a whole. This system-level integration is typically implied by the declaration of the aspects: any aspect declared in the system will be automatically integrated during execution. In this way, the base program can be *oblivious* to the aspects (that is, the base program does not need to know about the existence of the aspects) [6]. This obliviousness would allow new aspects to be developed and imposed on a base program after the base program is developed or even deployed. This capability may simplify the process of software evolution.

System-level aspect integration can best support the system-level aspectual decomposition. Using such a strategy, the decomposition must be identified and formulated during system design phase. Such a decomposition strategy seems mostly suitable for modularizing the system-scope reacting tasks, which are often required for implementing the system-wide crosscutting concerns, such as logging. However, as illustrated in Example 1, reacting tasks may exist in smaller scopes, like in the implementation of a particular method in a class. In such situation, a system-level decomposition strategy may be inappropriate.

EOS [17] and EOS-U [18] allow instance-level integration. In these techniques, a program can create and propagate aspect instances pretty much in the same way as that for class instances. This aspect instance can then be integrated with other instances by advising the actions performed by the advisee instances. The advisee instances may be added or removed dynamically during program execution. The instance-level integration avoids the notion of base program. It sacrifices the benefits of the obliviousness for the better control of the integration between the aspect instances and the rest of the program. It supports instance-level aspectual decomposition that can be identified and formulated at object-design level. This decomposition strategy seems suitable for instance-level reacting tasks that may often required for component integration where invariants must be maintained between several components.

Instance-level aspectual decomposition requires both the reacting tasks and the driving tasks in a reactive collaborating behavior to be captured by instances created from various modules in a program. Therefore, applying such a decomposition strategy too frequently may result in a large number of "fragmented" modules. In addition, if the reacting tasks and the driving tasks are closely interacting with each other (e.g., the worklist updating task and node processing task in `prop()` in Figure 1), capturing them with separate modules may result in a scattered and fragmented implementation of a cohesive functionality. For these reasons, new aspect integration mechanisms may be needed to to support finer grain aspectual behavior decomposition.

4 A Critical Analysis of Aspect Advising

The actions captured in an aspect must be performed through advising. Existing AOP techniques provide three different kinds of advising schemes: whole program advising, control-flow-level advising, and instance-level advising. These advising schemes differ in the restrictions on the execution contexts under which the advising can take effect. In the whole-program advising scheme, there is no restriction on the execution contexts under which the aspect instance can advise actions. In the control-flow-level advising scheme, each aspect instance will be associated with a control-flow context that represents a specific period of time during program execution; and the aspect instance can only advise actions performed within this period of time. In the instance-level advising, each aspect instance will be associated with one or more class (or classpect) instances; and the aspect instance can only advise the actions performed by the associated instances or the actions performed on the associated instances. AspectJ supports all three advising schemes. EOS and EOS-U support only whole program advising and instance-level advising.

One major limitation with the existing advising schemes is that they do not distinguish the execution context of an aspect instance from the execution context under which the advising of this aspect instance may take effect. That is, it does not distinguish the actions that are performed for the reacting task captured by the aspect instance from actions performed for a corresponding driving task. If the same kind of action may be performed by these two different tasks, the aspect instance may accidentally advise its own actions and thus may cause unexpected behaviors. For example, the tracing aspect defined in Figure 2(a) may cause infinite recursion: when `dumpState()` method is called, the advice code will be invoked recursively. AspectJ provides several different kinds of pointcuts (e.g., control-flow pointcuts, advice execution-related pointcuts, program text-based pointcut) that can be used in a pointcut expression to include or to exclude joinpoints in certain execution contexts. These pointcuts may be used to mitigate this problem. For example, the tracing aspect defined in Figure 2(b) will not have the problem of infinite recursion.[5] As we can see, such a solution is low-level, tedious, and indirect.

---

[5] This solution is extracted from the "AspectJ Programming Guide" come with AspectJ.

```
aspect Tracing {
  before(): call(* *(..)) {
      p.dumpState();
  }
}
```

(a)

```
aspect Tracing {
 pointcut myAdvice():
      adviceexecution() && within(Tracing);
  before(): call(* *(..)) && !cflow(myAdvice) {
      p.dumpState();
  }
}
```

(b)

Figure 2. Aspects written in AspectJ.

Another major limitation with the existing advising schemes is their implicit invocation semantics. The implicit invocation differs from the explicit invocation in that the binding between a method/advice (the *callee*) and a point where the callee will be invoked is specified separately from the program text (the *caller*) that contains the invoking point. That is, the caller does not have any indication of the possible invocation of the callee. This changes the conceptual relationship between the caller and the callee. In the explicit invocation, the callee is used by the caller to implement its responsibilities. However, in the implicit invocation, the callee now is used to modify the responsibility of the caller. From this perspective, the conceptual role of an aspect instance can be viewed as a behavior modifier.

The use of implicit invocation creates complications in maintaining contract that is imposed on the caller. In explicit invocation, the effect of the callee has been considered in the contract for the caller. However, in implicit invocation, this is not the case. Thus, modifying the behavior of the caller by invoking the callee at various joinpoints may force the caller to violate the predefined contract. Because there is no trace in the caller about the possible invocation of the callee, such a violation may be very difficult to detect. When aspects are used for capturing reacting tasks that are tightly coupled with the corresponding driving tasks in reactive collaborating behaviors, such violation may often occur and even skip the detection of software developers. In this situation, using aspects may adversely affect software quality and programming productivities.

The implicit invocation semantics of the advising schemes can cause further complications when an aspect instance can dynamically add or remove the associations with the instances that it can advise. Such a capability is provided by EOS and EOS-U. As we mentioned, an aspect instance is a behavior modifier. When an instance is associated to an aspect instance, its behavior will be modified by the aspect instance. The effect of such modification will be maintained until the association is removed. If such associations can be added or removed dynamically during the progress of the execution, then to reason about the program, the software developers must be able to track such a behavior modification-side-effect. In the presence of aliasing and heap-allocated instances, tracking the behavior modification-side-effect would be challenging.

The implicit invocation semantics of the advising scheme also complicates the use of exceptions in the program. As we mentioned, aspects are used to capture the reacting tasks in reactive collaborating behaviors. If needed, an aspect should be able to use reacting-task-specific types of exceptions to communicate with the execution context that contains the reactive collaborating behavior. However, with the existing advising schemes, this may not be possible. Programming languages like AspectJ rely on the stack for propagating the exceptions and searching for the appropriate handlers. Using this mechanism, an exception thrown by an action captured by an advice will be injected into the method (the caller) that contains the current joinpoint. If the type of the exception is checked and this type is not declared in the signature of the caller, then throwing such an exception is prohibited. Even if the exception is allowed to be thrown by the advice, injecting a reacting-task-specific exception in the advice may interfere with the exception handling in the caller since the caller is developed without knowing the existence of the advice. We refer to this problem as the *exception injection* problem.

5 The Statement-Level Aspectual Threads

This section presents a new AOP technique that can overcome some of the limitations that we discussed in the previous sections. More details can be found in [13].

5.1 An Overview

To avoid some of the problems with the existing AOP techniques, we propose to use threads to capture the reacting tasks and the driving tasks in a reactive collaborating behavior. A thread conceptually is an active execution entity. During program execution, a set of threads can progress concurrently. Each thread maintains its own state separately from other threads. Threads can share data. They can also coordinate their actions. Threads seem a natural fit for describing collaborating tasks: in this situation, each task can be captured by a thread, and the collaborating protocol can be implemented with thread coordination mechanisms.

Unfortunately, threads and thread integration mechanisms provided by existing programming languages may be too complicated to use in many situations. First, threads are typically coordinated through a set of thread communication and synchronization primitives (e.g., monitors, semaphores). With these primitives, the statements that specify the collaborating protocol for a set of tasks may be scattered in various threads that capture these tasks. These statements often tangle with statements used for other purposes. Therefore, it may be challenging for understanding and reasoning about the interactions among the collaborating tasks. Second, threads provided by many programming languages may be scheduled nondeterministically. While this scheduling scheme may improve the responsiveness and/or performance of the software system, it complicates the programming model. For example, to ensure proper sharing of resources, complex locking mechanisms may need to be introduced. The nondeterministic scheduling also complicates the software verification and understanding: the software developers now have to consider the effects of all the possible orders in which the actions get scheduled. For these reasons, programs that use this kind of threads often contain very subtle errors that are hard to detect, to repeat, and to fix. As pointed out by Ousterhout [16], using this kind of threads, "even for experts, development is painful".

In our technique, we introduce program constructs that can simplify the use of threads in describing reactive collaborating behaviors. Our technique supports a binary decomposition strategy. For each reactive collaborating behavior that is identified during design, the software developers can identify one reacting task from the collaborating behavior and capture this task with a *reacting* thread; the software developers can then capture the other tasks for the collaborating behavior into a *driving* thread. This binary decomposition can simplify the relationship among different threads.

Our technique uses an observation-based mechanism for integrating a reacting thread with the corresponding driving thread. Using this integration mechanism, the driving thread can execute without explicit coordination with the reacting thread; and the reacting thread will schedule its actions according to the execution events that it observes from the driving thread. This arrangement allows the specification of the driving thread to be completely free of the coordinating codes. The arrangement also allows the specification of the reacting thread to be organized around event-action bindings that explicitly specifies the collaborating protocol among the two threads. In this way, we can separate the specification of the two threads, and yet can avoid scattering the codes for implementing the collaborating protocol. From the above description, we can see that the observing capability of a reacting thread is quite similar to that of an aspect instance. Therefore, we also refer to a reacting thread as an *aspectual* thread.

Our technique also supports a hierarchical decomposition strategy. That is, within a reacting thread or a driving thread, the software developers can further identify reactive collaborating behaviors and use child threads to captures the tasks in such behaviors. To support this decomposition strategy, our technique uses a statement-level thread integration model. In this model, an aspectual thread and its corresponding driving thread will be specified in two different blocks within an *observation* statement. This model is very similar to the `cobegin/coend`-like constructs available in several existing multi-threaded programming languages. Using this model would allow the aspectual behavior decomposition strategies to be applied within a method.

Our technique uses a deterministic execution model for the threads. In our technique, only one thread can be executing at a particular point of time. When the program execution of a thread reaches the beginning of an observation statement, an aspectual child thread and a driving child thread will be created from their specifications. After that, the execution of the current thread will be suspended and the driving child thread will start its execution. During the execution of this driving thread, when an execution event that is interesting to the aspectual thread occurs, the driving thread will be suspended at this point of time to allow the aspectual thread to perform the response action. After the aspectual thread finishes the execution of the response action, it will enter a waiting state and the driving thread can then resume its execution. This event observation/response can repeat until one of this pair of threads terminates. In this case, the other thread will also terminate and the execution of the parent thread will be resumed. This execution model of thread is deterministic.[6] Therefore, programming with this model can avoid many subtle problems that may be encountered when programming with traditional thread model.

5.2 An Example

We use an example to illustrate how to capture the reactive collaborating behaviors with the observation statements. Figure 3 shows an alternative implementation for the algorithm shown in Figure 1. This implementation uses an observation statement for capturing the collaborating behavior that involves the worklist maintaining and the node processing. In the observation statement (the statement starting with keyword `observe`), the node processing is captured in the driving thread defined in the Java block following the keyword `within`; and the worklist updating is captured in the aspectual thread defined in the block between keywords `observe` and `within`. In the block that captures the aspectual thread, a *guarded command* (the statement starting with keyword `on`) specifies the type of

---

[6] The execution model we discuss here is very similar to that of the coroutines. However, since our threads are defined at statement-level, instead of the procedure level, we avoid using the name coroutine.

```
class Node {
   Set ents;
   void addE(E e) {
     if( !isIn(e) )
        ents.add(e);
   }
   void process() {
     // as described in (a)
   }
   boolean isIn(E e) { ... }
   // other methods and fields
}
...
void prop(Node root, E ee) {
   root.addE(ee);
   WList wlist = new WList(root);
   try {
  L: observe {{
       on enter Node.addE(E e): {
         if(!receiver.isIn(e))
            wlist.add(receiver);
            // may raise WListExcpt
       }
     }} within {
       while( !wlist.empty() ) {
          Node n = wlist.getNext();
          n.process();
       }
     }
   } catch(WListExcpt e) { ... }
}
```

Figure 3. The implementation of a worklist algorithm with observation statement.

event to be observed and the actions to be performed when such an event occurs. The observation statement states that, during the node processing, whenever `Node.addE()` is invoked to add a new entity to a node, this node (referenced by the implicit variable `receiver`) 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 within `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. Also note that, in the new implementation, the exception that may be raised in `WList.add()` now can be propagated directly to `prop()`. Thus, none of the signatures for the methods of `Node` need to declare the throwing of such an exception. Due to these improvements, if the software developers decide to change the representation of the worklist (e.g., use a Vector instead of a WList), or to change the information that will be stored in the worklist, this modification will be confined within `prop()`. Thus, this maintenance task would be much easier.

5.3 Single-Stack Thread Implementation

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. However, threads created by the observation statements have a simpler runtime model. This run time model allows us to implement these threads with a single stack.

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 occurance of events. An executing thead 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 a particular 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 finish executing the response action and enter the waiting state. In both cases, a suspended thread can reclaim the execution control from another thread only if the other thread is terminated or has entered the waiting state. Thus, all method invocations of the other thread should have been returned. This characteristic allows different threads to be run on the same stack. In our implemtnation, 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 Machine. The approach can also reduce the time and space overhead required for coordinating the threads created by the observation statements.

Like the aspect integration in other AOP techniques, coordinating the actions in different threads in our technique may require runtime support for event observation and dispatching. This runtime support can be implemented by inserting probing statements at appropriate locations in the program to detect the execution events that may be interesting to some aspects. The insertion of the probing statements can be done by a compiler or a binary code rewriter. More details on the design issues related to our implementation of the observation statements can be found in Reference [13].

5.4 The Discussions

We propose the statement-level aspectual threads as a new form of aspects. As shown by the example in Figure 3, they may be used to form new decomposition strategies for avoiding code scattering and tangling problems that may not be accomplished with existing AOP techniques. The aspectual threads also present several advantages over the aspect instances proposed by existing AOP techniques.

First, aspectual threads present a cleaner programming model than aspect instances. Different threads maintain their own states. Therefore, unlike an aspect instance that is viewed as a behavior modifier, an aspectual thread is an independent reactive component whose responsibility is to process execution events that occur in the corresponding driving thread. This change of perspective subtly changes the way in which the program can be reasoned about. In existing AOP techniques, an advice in an aspect instance must be applied to the joinpoints to understand its effect on the program states. It also requires the software developers to track the behavior modification-side-effect when reasoning about the program. In contrast, in our technique, the effect of a response action in an aspectual thread will be reflected in the state of the thread and the data shared by different threads. As long as the sequence of events observed by an aspectual thread and the data sharing protocol are well understood, the effect of the aspectual thread on the program states can be reasoned about by examining the specification of the aspectual thread. This approach can better support modular reasoning.

Second, the coordination mechanism between a driving thread and an aspectual thread provides a less tangled execution model than the advising mechanism for aspect instances. Threads maintain their own execution contexts. With this separation, an aspectual thread will observe only the events occurred under the execution context of the corresponding driving thread. The aspectual thread will never respond to events that occur under its own execution context. Thus, it can avoid the recursion problem that may arise with existing advising mechanisms. In addition, the uncaught exceptions raised in an aspectual thread and its corresponding driving thread will be propagated to their parent thread separately. For example, in Figure 3, an exception raised by `wlist.add()` in the aspectual thread will be propagated to `prop()` without interfering with the exception handling of the driving thread. In a sense, an aspectual thread is seaprated from the corresponding driving thread not only in specification, but also in program execution. Therefore, using aspectual threads, we can avoid the exception injection problem discussed in Section 4.

Third, the statement-level integration of the aspectual thread is more manageable than the existing instance-level integration of the aspect instances. Using statement-level integration, the life time of a thread is restricted within the scope of an observation statement. Once the observation statement terminates, the thread will be removed from the system. Therefore, the effect of an aspectual thread is well-contained. In contrast, using instance-level integration, the life time of an aspect instance and the life time of an association between this instance and any of its advisee instances are determined by the statements that are scattered in the program. In this case, reasoning about the integration of aspect instances in a program is quite challenging.

Of course, to gain these advantages, the program constructs that we introduce may not be as powerful as the existing AOP program constructs. First, unlike an aspect instance, an aspectual thread cannot substitute or remove behaviors in the driving thread. This kind of expressiveness may be handy from time to time, especially when the aspects are used to evolve an existing program. Second, the statement-level integration may not be convenient enough in some situations. For example, what if the driving task in a reactive collaborating behavior is captured by an object? In this case, we may need to introduce instance-level integration. For these reasons, we do not view the statement-level aspectual threads as the only and final answer for capturing the reactive collaborating behaviors. More experiences and experiments are required for us to understand deeper the nature of the reactive collaborating behaviors and the advantages/limitations of the existing (AOP or non-AOP) techniques in capturing such behaviors. Such understandings may lead to the development of program constructs that can conveniently modularize the collaborating behaviors.

Aspectual threads can be used together with normal Java threads. In this case, each normal Java thread will be a root thread that may create child aspectual threads and child driving threads. These child threads are considered as a integral part of the root thread. Therefore, a child aspectual thread cannot observe actions performed by other normal Java threads. This is yet another feature that distinguishes an aspectual thread from an aspect instance: in an aspect instance, an advice can take effect whenever an associated joinpoint is reached in the base program. This feature may simplify the program reasoning in when the aspectual threads are used by normal Java threads.

6 Conclusions and Future Work

In this paper, we propose that aspects should be viewed as program constructs that support better decomposition and modularization of reactive collaborating behaviors. Because program behaviors are identified and analyzed at various design levels, this vision may allow software developers to systematically identify design situations where aspects may be useful. We also identify several limitations with existing AOP techniques in supporting this vision. We finally present an overview of statement-level aspectual threads that serve as an example to illustrate how some of these limitations can be overcome.

The statement-level aspectual threads present an interesting direction in dealing with code scattering and tangling problems that may present in systems implemented with traditional programming paradigm. Much remains to be done. In the theoretical perspective, we will further investigate the interactions among the different tasks in the collaborating behaviors. Such investigation should allow us to classify these interactions and to understand the design problems in capturing these behaviors in a program. Such understanding may allow us to develop more effective decomposition strategies for such behaviors and program constructs to support such strategies. In the practical perspective, we will implement and evaluate the language constructs that we develop. We will use them in realistic software projects to further understand the power and the limitations of these language constructs. Such understanding may lead to further improvement of the constructs.

References

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

[2] L. Bergmans and M. Aksit. Composing crosscutting concerns using composition filters. *Communications of the AC*, 44(10):51–57, 2001.

[3] 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.

[4] 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.

[5] 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.

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

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

[8] D. Garlan and D. Notkin. Formalizing design spaces: Implicit invocation mechanisms. In *VDM'91: Formal Software Development Methods*, 1991.

[9] 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.

[10] 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.

[11] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. ECOOP*, pages 220–242, 1997.

[12] R. Laddad. *AspectJ in Action*. Manning Publications Co., 2003.

[13] D. Liang. Modularizing reactive collaborating behaviors. Technical report, University of Minnesota, 2004.

[14] K. Lieberherr, D. Orleans, and J. Ovlinger. Aspect-oriented programming with adaptive methods. *Communications of ACM*, 44(10):39–41, 2001.

[15] H. Ossher and P. Tarr. Using multidimensional separation of concerns to (re)shape evolving software. *Communications of ACM*, 44:43–49, 2001.

[16] J. K. Ousterhout. Why threads are a bad idea (for most purposes). Invited talk at the 1996 USENIX Technical conference, 1996.

[17] H. Rajan and K. Sullivan. Eos: instance-level aspects for integrated system design. In *Proceedings of the 9th European software engineering conference held jointly with 10th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 297–306, 2003.

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

[19] P. Tarr, H. Ossher, W. Harrison, and S. Sutton. N degrees of separation: Multi-dimensional separation of concerns. In *Proceedings of the International Conference on Software Engineering (ICSE'99)*, 1999.

[20] 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.