Specifying Reusable Aspects

Neelam Soundarajan
Ohio State University
neelam@cse.ohio-state.edu

Raffi Khatchadourian
Ohio State University
khatchad@cse.ohio-state.edu

Abstract

Aspect-Oriented Programming enables developers to manage, in a more modular fashion, implementations of crosscutting concerns that might be scattered or tangled if aspect-oriented techniques were not utilized. Our interest in this paper is in considering techniques for specifying precise properties of aspects. In particular, we are interested in specifying *reusable* aspects; i.e., aspects that correspond to crosscutting concerns that occur in many systems. These *abstract* aspects can be reused in various systems where a particular concern is applicable. Although there has been work on issues related to reasoning about aspects and the behaviors of aspect-oriented systems, specifying reusable abstract aspects seems to have received little attention.

We adopt the following approach to specifying reusable aspects and how they are specialized in individual systems. We specify a reusable aspect in terms of a *contract*; the contract is written in terms of *abstraction concepts* that will correspond to portions that are left abstract in the reusable aspect. This contract will be *specialized* by a *subcontract* corresponding to the subaspect that defines how the abstract aspect is specialized for use in a given system. While the aspect contract is applicable to all systems built using the reusable aspect, the contract plus the subcontract define the behavior corresponding to the given system. We illustrate the approach by applying it to a simple example.

1 Introduction

Aspect-Oriented Programming (AOP) allows developers to manage, in a more modular fashion, implementations of *crosscutting concerns* (CCCs) that might be scattered among multiple modules or tangled with other unrelated code if AO techniques were not utilized. Our goal in this paper is to consider formal specifications of *reusable* aspects. Previous work on specifications of aspects has focused primarily on aspects that are defined for particular systems with the goal of modifying or enriching the behaviors of these systems in specific ways; the main facet of much of this work, some of which we will summarize in the next section, has been on such problems as ensuring that the existing specifications of the *base-code* (i.e., the underlying program) are not compromised or invalidated as a result of the actions of the added aspects. While such aspects are undoubtedly very useful, we believe the power of the AO approach can be of particular value in the case of reusable aspects that are designed to address commonly occurring CCCs, and that can then be used in various systems as needed. To ensure that this is done effectively, we need appropriate, precise specifications of such aspects that can then be used to understand the behavior of the systems in question. Developing ways to write specifications for reusable aspects is our main goal.

Before considering the details of how such specifications can be written, it may be useful to briefly summarize some of the main technical terms used in AO programs and systems. For convenience, we will use the syntax and terminology of *AspectJ* [18] although our approach does not depend on the details of the particular language[1]. A *join point* identifies a specific point in the execution of a program such as a call to a particular method. A *pointcut* groups together a set of join points that we want to treat in a uniform manner, for example all the calls to a particular method. The *advice* associated with a pointcut specifies the code that should be executed when control reaches any of the join points that match the pointcut. The pointcut enables us to collect *context*, for example, the object on which the method in

[1] At the same time, we should also stress that we consider only a small portion of *AspectJ*. The language has many powerful features that we do not consider since our primary interest is on precise specifications of reusable aspects; hence we restrict attention to a small set of features that enables us to express such aspects.

question was applied. The collected information can then be used in the advice code. There are three kinds of advice, before, after, and around. Suppose a join point is a method call. The before advice, if any, that applies at this join point is executed before the method is executed. The after advice, if any, is executed after the method is executed. And the around advice, if any, is executed in place of the method call; the around advice may include one or more calls to "proceed," which invokes the original method.

An *aspect*, typically, defines one or more pointcuts and the items of advice applicable at each of them. One possibility, especially important from our point of view, is that a pointcut may be specified to be *abstract*, to be defined in the *subaspect*(s). Although advice cannot be specified to be abstract, having the advice invoke a method of the aspect and specifying the method to be abstract can achieve the same effect. If any part of an aspect is abstract, the aspect itself is considered abstract (and must be so indicated in its header). Abstract aspects are key to creating reusable aspects since the abstract aspect can define the portions that apply to *all* systems that use the aspect, with system-specific concrete subaspects being used to specialize the abstract aspect as needed for the individual systems.

In general, aspects are used to represent *crosscutting concerns* (CCCs); i.e., concerns involving multiple objects in the system. In standard OO versions of systems that involve such concerns, the code for dealing with the concern will typically be scattered across the multiple classes corresponding to these objects (or unnaturally forced into one of the classes); and this code may also be tangled with code that has nothing to do with the particular concern. It was this potential for the code for this CCC being scattered and tangled in this manner that was a primary force in the development of the AO approach. The introduction of constructs such as pointcuts and advice allow the CCC to be encoded in one place as an aspect. In the case of CCCs that appear in multiple systems, it would clearly be advantageous, as noted above, to code the common portions of these CCCs in the form of suitable abstract aspects.

Other authors have stressed the importance of accounting for reusable aspects of this kind early in the design of systems as well. Baniassad, Clarke, and Walker [2, 6, 7] introduce the idea of *composition patterns*, which can be used to describe, at the design-level, various CCCs. They also introduce ThemeUML, an extension to UML, to specify the structure and some facets of the behaviors, in a manner that is compatible with UML. They discuss how these can be translated in a natural manner into HyperJ [21] and AspectJ. In [7], Clarke and Walker present a simple example of crosscutting behavior that may arise in multiple systems. They show how this concern can be represented in their ThemeUML and how it can be encoded as an abstract aspect in AspectJ. They also present a simple example system in which this CCC appears and show how a subaspect can be defined that specializes the abstract aspect.

Suppose *B* is a particular CCC that appears in many systems. Suppose a set of abstract aspects *AB* encodes the parts of *B* that are common to these systems. And suppose *SB* is a corresponding set of concrete aspects that specialize *AB* in a manner appropriate to the needs of a particular system *S*. The key questions we wish to address then are as follows. How do we precisely specify *AB* and *SB*? How do we ensure that any requirements that the specification of *AB* may impose on *S/SB* are satisfied? We will present one possible set of answers to these questions. In our approach, a *contract* will specify information about the abstract aspects *AB*; the requirements and invariants specified in the contract will apply to all systems built using *AB*. The contract will be *specialized* by a *subcontract* corresponding to *SB*. In other words, the particular manner in which *AB* is specialized in the system *S* will be specified by the corresponding subcontract while the information in the contract for *AB* will be common to all systems built using *AB*. We will illustrate the approach by applying it to the abstract aspect presented by [7]; we will develop the contract for this aspect. We will then consider the subcontract for a particular system that uses this abstract aspect.

Other authors (see, for example, [3, 20, 26]) have also worked on *early aspects*, i.e., aspects that are considered early in the system design (rather than being added as afterthoughts). To the extent that the concern represented by an early aspect appears in multiple systems, it would be appropriate to encode, in an abstract aspect, portions of this concern that are common to these systems rather than being coded

each time from scratch; this abstract aspect can then be specialized for use in the individual systems by defining appropriate subaspects. Our approach will be of value for such cases.

In the next section, we will consider related work. We will briefly discuss some of the approaches that have been developed for reasoning about the behavior of aspect-oriented systems, as well as some of the work on early aspects. In the third section, we develop our contract/subcontract-based approach to specifying reusable aspects and the concrete aspects corresponding to particular systems; we use a running example to illustrate the approach. In Section 4, we summarize our approach and consider directions for future work.

2 Background and Related Work

As noted in Section 1, much of the work on reasoning about AO systems has attempted to address such questions as how to ensure that the existing specifications of the *base-code* of a system *S* are not invalidated as a result of the action of aspects that are added to *S*. Let us consider this in more detail. Suppose *C* is a class whose methods have been specified using standard pre-/post-conditions. Suppose an aspect *A* is added to the system to which *C* belongs. Suppose, given the pointcuts defined in *A*, that some of the advice in *A* is applicable at various join points in methods of *C*. How can we be sure that the existing pre-/post-condition specifications of the methods of *C* that were obtained without accounting for *A* will remain valid under the action of the advice in *A*? Alternately, if the behavior of the methods of *C* will be modified in important ways, how do we arrive at these new behaviors without reanalyzing the bodies of the methods?[2] These are the questions that much of the previous work on specifying aspects has focused on. Thus, Dantas and Walker [10] introduce the idea of *harmless* advice, i.e., advice that does not modify the behavior of the base-code. Clifton and Leavens [9] generalize this by introducing the notion of *spectator* advice; i.e., the method code plus the interwoven advice still behaves in a way that satisfies the original specification of the method in question. They also consider more general type of advice, called *assistants*, which might not satisfy the requirements for *spectator* advice; and consider how the effect of such advice can be accounted for with minimal reanalysis.

Similarly, in our earlier work [17], we considered how to provide extra information–beyond that typically contained in post-conditions of methods–that can be used to arrive at the "richer" behavior of the method resulting from the execution of applicable advice without reanalysis of the base-code. This information concerns the state of the object at various key points in the method body, these points being the potential join points at which advice might apply; this information can be combined with the effect of the advice code to arrive at the richer behavior of the method. The specifications in [17] can also impose conditions on the behavior of the advice code. Katz and Katz [16] consider similar *rely-guarantee* specifications of advice code; in particular, they consider the possibility that the combined effect of two or more items of advice might cause the conditions imposed on the behavior of advice being violated even if each item of advice *individually* satisfies the conditions.

Other authors have proposed ways to minimize the effect of aspects on the behavior of base-code and/or to make it easier to identify and understand the effects. For example, Aldrich [1] proposes the notion of *Open Modules* in which aspects are only allowed to provide advice that applies at join points that the base-code has explicitly "exposed". Griswold *et al.* [12] introduce the notion of *XPIs* (crosscutting interfaces), which abstract a crosscutting subset of the base-code that correspond to points that are likely to be of interest to an aspect developer. XPIs also include an (explicit or implicit) specification of the condition that will hold when control reaches any of the matching points. The aspect can only provide advice at the XPIs exposed by the base-code; conversely, the base-code designer is obliged not

[2] If we were willing to reanalyze the method bodies, we could, during this reanalysis, consider the behavior of the bodies with appropriate advice code woven in. The point of modularity is, of course, to avoid such reanalysis.

to make changes to the base-code in ways that would violate the specified conditions. Such restrictions considerably simplify the problem of considering the behavioral effect of the aspect on the base-code.

Jacobson and Ng [15] consider the effect of aspects on individual use cases. The advice corresponding to the aspect is specified as an *extension* use case. But since the base use case does not mention the aspect, it is necessary to separately specify the *extension points*, i.e., the points where the extension applies; this is rather similar to defining pointcuts where a given piece of advice should apply. Sousa *et al.* [25] propose techniques for incorporating crosscutting behavior into the design of the system's use cases. They also stress that standard *includes* and *extension* relations between use cases are not adequate and present a new relation, *crosscuts*, to capture this; it is used to specify the interaction of the use case for the advice in this aspect with each of the affected base use cases.

Some of the work on *early aspects* takes a different approach. Rather than treating aspects as afterthoughts that might modify the behavior of the base-code in possibly undesirable ways, the early aspects work attempts to account for CCCs and the corresponding aspects early in the system lifecycle, starting from the *requirements* phase. Rashid *et al.* [22] and Chitchyan *et al.* [5] consider the question of modularizing and composing aspectual requirements. They use a common abstraction, *concern*, to represent both crosscutting and non-crosscutting elements. Concerns specify requirements and can be organized hierarchically. *Composition* elements specify how the individual concerns are assembled and the desired interactions among them. Similarly, Mikkonen [20] discusses the need for accounting for aspects in the *architecture* of a system. He investigates possible ways in which the relation between aspects and objects/classes can be considered in the system's specification. Some authors focus on aspects corresponding to non-functional requirements such as persistence, distribution, etc. Thus, Grundy [13] considers how the requirements dictated by these concerns may be decomposed into "aspect details" and how these, in conjunction with the functional requirements, may be refined into detailed component designs. In a sense, Grundy's approach can be considered as the analog of "weaving" at the design stage.

The importance of considering aspects early in the lifecycle is also the motivation of some other authors who focus on the *design* stage. Clemente *et al.* [8] argue that while there are good tools for AO at the programming level, the lack of suitable designing and modeling methodologies for AO seriously hampers its effective use. They offer two alternatives; first, an extension of UML with an *aspect profile*; second, the introduction of new primitives in architectural description languages to support aspect definition and composition. There is also an important relation between such aspects and *design patterns* [11]. Hannemann and Kiczales [14] show how a number of GoF patterns can be implemented as abstract aspects. Our approach borrows ideas from our earlier work [23, 24] on specifying design patterns. One key difference between design patterns and (reusable) aspects is that the former expresses, in a sense, design *intent*, whereas the latter is actual code to be processed by the AO compiler/run time system.

Zhao and Rinard [27] introduce a specification notation, *Pipa*, as an extension of *JML* [4], to specify aspects. Their focus is on specifying the behavior of various items of advice that may appear in an aspect using an approach similar to that used in JML to specify behavior of methods of classes using *model* variables where necessary. They also consider the notion of invariants associated with aspects. Furthermore, they discuss the possibility of *weaving* together the (original) specifications of methods with the specifications of applicable items of advice to arrive at the final specifications of the methods; their intent is that these resulting specifications can be checked using existing JML tools. Although Zhao and Rinard consider abstract aspects, the ways in which they allow an aspect to be abstract are rather limited. Consequently, their specifications allow for only limited kinds of specializations by the subaspects. As we will see, to realize the full power of reusable aspects, subaspects must be allowed a rich set of specializations. AspectJ does allow this, and our formalism, via the notion of *abstraction concepts*, represents this in specifications of abstract aspects.

3 Specifying Reusable Aspects

Suppose *B* is a particular CCC that appears in many systems; suppose *S* is such a system. Suppose *AB* is an abstract aspect that encodes the portion of *B* that is common to all systems, including *S*, that involve *B*. Then *AB* will consist of some portions that are flagged as *abstract* and other (concrete) portions that are defined in *AB*. For any system *S* in which *B* appears as a concern, we will define a *subaspect SB*. *SB* will be defined so that it *specializes AB* in a way that is appropriate to *S*. The questions we have to address are, how do we specify *AB* and, given *SB*, how do we specialize that specification so that we obtain the behavior that the aspect brings to *S*?

In our approach, we will specify the behavior of *AB* in terms of a *contract*. What is specified in the contract will apply to *all* systems built using *AB*. For each subaspect *SB* of *AB*, we will obtain a specialization, which we will call a *subcontract*, of *AB*'s contract that will correspond to the specialized behavior of the aspect/subaspect. A key question that we have to consider is how, in *AB*'s contract, do we account for the portions of *AB* that are *abstract*? These pieces are critical because, although they are not defined in *AB*, they are, in general, referred to by other parts of *AB*. We will use what we will call *abstraction concepts* to represent these pieces in *AB*'s contract. The abstraction concepts, as the name suggests, will not be defined as part of the contract but they will be used in specifying various portions of the contract. The actual definitions of the abstraction concepts, corresponding to a particular subaspect, will be provided as part of the specification of the subaspect. These definitions can then be combined with the aspect contract to obtain the subcontract corresponding to this aspect/subaspect. We will illustrate these ideas using an example based on one in [7].

3.1 Reusable Aspect Example: Observing Protocol

```
1  public abstract aspect Observing {
2    // type declarations
3    interface Subject { }
4    interface Observer {
5      public void update();  }
6    // introductions
7    Vector Subject.observers;
8    Vector Observer.subjects;
9    private void Subject.notify() {
10     // invoke update() on all elements in Subject.observers }
```

Figure 1: Observing Aspect (part 1)

Consider the *Observing* aspect defined in Figs. 1 and 2. This aspect encodes the standard *observer protocol*. `Subject` and `Observer` are the two required interfaces. The latter is required (lines 4–5) to define the `update()` method. But `notify()` is not part of `Subject` (as it often is in discussions of the related *Observer* design pattern). It is instead defined in the aspect (lines 9–10) using information about the current `observers` also maintained (line 7) by the aspect; `subjects` (line 8) similarly allows an `observer` to keep track of multiple subjects it might be observing. It is appropriate to define `notify()` as part of the aspect (rather than as a method of `Subject`) because it involves a CCC.

Pointcuts and advice are defined in Fig. 2. `subjectModified()` (line 13) corresponds to points where the state of the `subject` has been changed. But "changed" is not intended to be taken literally. Instead, what "changed" means will depend on the system in question. More precisely, it will depend on the kinds of changes in the `subject` state that will be of interest to potential `observers` in the particular system. Thus, while in one system we may have `observers` that are interested in every part of the state of the `subject`, in another the `observers` may be interested only in changes in certain portions of the `subject` state. In order to allow for such differences, we specify this pointcut to be

```
12  // pointcuts
13  abstract pointcut subjectModified(Subject s);
14  abstract pointcut attach(Observer o, Subject s);
15  ... detach is similar ...
16  // advice
17  after(Subject s):subjectModified(s){ s.notify(); }
18  after(Observer o, Subject s):attach(o, s){ s.observers.add(o); }
19  ... similar for detach ...
```

Figure 2: Observing Aspect (part 2)

abstract, with the actual definitions being required to be provided in the subaspects corresponding to the individual systems. But the action to be taken when control reaches a join point that is in this pointcut is *not* abstract – it is the advice (line 17) that calls `notify()` (which, as defined in Fig. 1, invokes `update()` on all attached `observers`).

The `attach()`/`detach()` pointcuts correspond to attaching/detaching an `observer` are also abstract. Alternatively, we could have included specific methods in the `Subject` and `Observer` interfaces corresponding to these actions. But the approach of treating these as abstract pointcuts allows for a greater degree of generality allowing different systems to handle attaching and detaching of `observers` in their own particular ways. While these pointcuts are abstract, the actions to be taken when control reaches a join point that matches either of these pointcuts, as in the case of the `subjectModified()` pointcut, are not abstract; they are defined in the aspect (lines 18–19). The advice bodies that will execute when control reaches any join point matched by these pointcuts add and remove the `observer` from the variable `observers` introduced into `Subject` (line 7), respectively.[3]

```
20  // Abstraction concepts
21  /*@ abstract boolean Modified(Subject s1, Subject s2);@*/
22  /*@ abstract boolean Consistent(Subject s1, Observer o1);@*/
23
24  /*@ abstract pointcut subjectOther(Subject s); @*/
25  /*@ after(Subject s):subjectOther(Subject s){ } @*/
26
27  /*@ constraint: [Consistent(s1,o1) ∧ (¬Modified(s1,s2))]  @*/
28  /*@               ⇒ Consistent(s2,o1)                     @*/
29  /*@ constraint: [Subject.Methods == subjectModified ∪ subjectOther] @*/
30  /*@             ∧ [subjectModified ∩ subjectOther == Φ] @*/
```

Figure 3: Contract for Observing Aspect (part 1)

Aspect Contracts. Let us now turn to the contract for the *Observing* aspect. The specification language we use for aspect contracts is inspired by JML [19] and Pipa [27]. We will not provide a formal syntax definition for the language; instead, we will explain various key parts of it as we consider the *Observing* contract, the first part of which appears in Fig. 3. Note that the lines that belong to the aspect contract (or, later, subcontract) are, following JML's convention, enclosed in the symbols "`/*@ ... @*/`".

Abstraction Concepts. The first part of the contract defines the *abstraction concepts* used in the contract. `Modified()` is a relation between two states of the `subject`. This concept is intended to represent the notion that the `subject` state has changed in such a way that the `observers` must be updated. In other words, if at some time during execution, the state of the `subject` is `s1`, a method is invoked on the `subject`, and, as a result of this method's execution, the state of the `subject` changes to `s2`, and the value of `Modified(s1,s2)` evaluates to *true*, then `update()` must be invoked on the attached `observers`. By specifying `Modified()` to be an abstraction concept in the contract of the

[3] Interestingly, there is a subtle bug in this code that we will return to later in the paper.

Observing aspect, the contract leaves it to the subaspect corresponding to an individual system (and the corresponding subcontract) to pin down which changes in the `subject` state require the `observers` to be updated and which do not.

`Consistent()`, the second abstraction concept (line 22), represents the notion of a state `o1` of an `observer` being *consistent* with the state `s1` of the `subject`. One simple notion of consistency would be for the `observer` to contain a *copy* of the current state of the `subject`; indeed, many discussions of the *observing* protocol give the impression that this is what is *required*. But in most situations, an `observer` is interested only in certain parts of the `subject` state and there is no need to save information about the rest of that state. All these possibilities are allowed for by defining `Consistent()` to be an abstraction concept in the aspect contract, to be defined in suitable ways in the subcontract.

Abstract Pointcuts. Line 24 introduces not an abstraction concept but, rather, an abstract pointcut. This pointcut is needed only in order to specify the aspect's *contract*. The reason is as follows. Suppose we build a system using the *Observing* aspect. Suppose in the corresponding subaspect, we provide a definition for `subjectModified()` that is *incomplete*; in other words, our definition of this pointcut is such that it does not include all of the methods that change the state of the `subject` in such a manner as to require updating of the attached `observers`. In this case, the system will fail to function correctly; i.e., some changes in the `subject` state (that are, in the sense of the `Modified()` concept, significant) will not be seen by the `observers`. The introduction of `subjectOther()` (and the associated contract) allows us to avoid this potential problem. Since this pointcut is introduced purely as part of the aspect contract, we flag it in the usual JML-style. The advice corresponding to the pointcut (line 25) is empty (as may be expected since we would not want contract-artifacts such as this pointcut and the associated advice to make changes in the states of any of the system objects).

Constraints. Although `Modified()` and `Consistent()` are abstraction concepts to be defined by the subaspects, as appropriate for the corresponding systems, the contract listed on lines (27–28) imposes a *constraint* on these definitions. To see the need for this constraint, consider the following situation. Suppose at some point in the execution of a particular system, the state of the subject is `s1`, the state of the (only) observer is `o1`, and `Consistent(s1,o1)` is satisfied, given the definition of `Consistent(s1,o1)` for this system. Suppose a method is now called on the subject and the execution of this method changes the subject state to `s2`. Suppose ¬`Modified(s1,s2)` is satisfied, again with the given definition of `Modified()`. This suggests and, as we will see below, the rest of the contract will allow, the join point for this method call not to be part of the `subjectModified()` pointcut. As a result, `notify()` will not be invoked and the observer state will remain as `o1`. But suppose `Consistent(s2,o1)` is *not* satisfied. Then the purpose of the *observing protocol*, which is to make sure that the states of the observers are consistent with the current state of the subject, will be violated – even though the protocol is being apparently observed! The source of the problem is that while the definition of `Modified()` for this system treats the change of the subject state from `s1` to `s2` to be something that the observer would not be interested in, the definition of `Consistent()` is such that the state `o1` of the observer is consistent with `s1` but not with `s2`. Clearly, such definitions of these concepts are mutually incompatible; the constraint in lines (27–28), which our formalism requires must be met by the subcontract corresponding to every system built using this aspect, ensures that such incompatibilities do not arise.

Specifying Advice. It is worth noting three points at this stage. First, AspectJ (and other AO languages) provide a number of types of join points. In practice, the most important one, and the one we will focus on, corresponds to method calls. The last two lines (29–30) of Fig. 3 require that the definitions of the two pointcuts corresponding to any subaspect of `Observing` must be such that each method of `Subject` is included in one of the two pointcuts; and their intersection must be empty. That is, for each method of `Subject`, we are required to include calls to it in exactly one of these pointcuts; if a method may or may not modify the state of the subject (depending, for example, on the values of some

parameters), it must be included in `subjectModified()`; otherwise, as we will see, it will violate a requirement in Fig. 4.

Second, as noted earlier, AspectJ provides three types of advice. In our discussion, we will focus on *after* advice; *before* advice can be treated in similar fashion as *after* advice; but *around* advice presents additional challenges that we will briefly consider in the next section. Third, in one respect, the execution of *after* advice is similar to that of the execution of a method call, the difference being that while the latter occurs as a result of an explicit call in the client code, the former occurs implicitly when another method, say `m()`, is called and the join point corresponding to calls to `m()` matches the pointcut to which the advice applies. Thus, in Pipa, specification of such advice (and of *before* advice) is quite similar to that of a method. In particular, the pre-condition of the advice is in terms of the state that exists when control reaches the start of the advice and the post-condition is in terms of that state and the state that exists when the execution of the advice finishes. But this is inadequate. In general, proper specification of the situation that holds when the advice finishes execution requires references also to the state that existed *at the start of the execution of* `m()` (as well as the states at the start and end of the advice execution); and the specification of the situation at the start of the advice, in general, requires references also the state at the start of `m()`. Hence in the post-condition of advice in our contracts, we will use the usual JML notation "`\old`" to refer to the state at the start of the execution of the advice; and a new notation, "`\oldM`", to refer to the state at the start of the execution of the *method* in question (`m()` in above discussion). `\oldM` can be used in the pre-condition of the advice for the same purpose.

```
31  after(Subject s):subjectOther(s);
32  /*@ pre: ¬Modified(\oldM s, s) @*/
33  /*@ post: ¬Modified(\old s, s) @*/
34
35  after(Subject s):subjectModified(s);
36  /*@ pre: true @*/
37  /*@ post: ¬Modified(\old s, s) @*/
38  /*@       ∧ [∀ob∈s.observers: Consistent(s, ob) ]@*/
39
40  /*@ invariant: [∀ob∈s.observers: Consistent(s, ob) ]@*/
```

Figure 4: Contract for Observing Aspect (part 2)

The pre-condition of the advice for the `subjectOther()` pointcut states that when any join point in this pointcut is reached, the subject state should be unmodified from what it was when the particular method call was made. Note that this doesn't require the current subject state to be *identical* to what it was when the method call was made. The requirement is simply that any changes in the state should be such that the observers do not care about it (in the sense provided by the definition of `Modified()` in the subaspect). Given our discussion thus far, this is a condition on all the methods defined in `Subject` other than those whose calls are in the `subjectModified` pointcut. The post-condition of the advice imposes a similar condition on the code of the advice. The empty advice defined in line 25 (Fig. 3) obviously satisfies this requirement. But it is interesting to note that we could change this code so that, for example, it counts the number of such calls – as long as this would not result in `Modified(\oldM s,s)` becoming *true*! Thus our contract suggests ways to make the reusable aspect more flexible.

The specification of advice for the `subjectModified` pointcut, in lines 36–38, is straightforward. It essentially requires that advice not modify the subject state (again in the sense of the `Modified()` concept); and requires the states of all the observers to be `consistent` with that of the subject. Assuming that the `update()` method defined in the `Observer` does ensure that the state of the observer in question becomes `consistent` with that of the `subject`, the advice defined in Fig. 2 (line 17), given the definition of `notify()` in Fig. 1 (lines 8–9), satisfies this requirement. But, again, the contract suggests that the advice can be made more flexible; for example, if some observers' states are already

consistent with the subject state, notify() does not need to invoke update() on them. The final part of the contract, line 40, tells us that if these requirements are met, the information in the various observers about the subject will be consistent with the current state of the subject.

3.2 Concrete Aspect Example: Library Observing

Clarke and Walker [7] present a simulated book library built using the *Observing* aspect. Instances of BookCopy act as subjects and instances of BookManager as observers. Part of the subcontract

```
39 aspect LibraryObserving extends Observing {
40  pointcut subjectModified(Subject copy):
41   target(copy) && args(..) && (execution(* BookCopy.bkBorrow(..)) ||
42    execution(* BookCopy.bkReturn(..)) );
43  pointcut subjectOther(Subject copy): ...all other methods...
44  definition of Modified(c1,c2): ...[c1.borrowed != c2.borrowed]...
45  definition of Consistent(c,m): ...m knows value of c.borrowed...
```

Figure 5: LibraryObserving subcontract (partial)

for this application appears in Fig. 5. The definition of the subjectModified pointcut (lines 40–42) specifies that calls to borrow or return the book match it. This is indeed as we would expect since these change the state of the book copy. The definition of Modified() (line 44) corresponds to this as well. The definition of Consistent() (line 45) states that the book manager has correct information about whether or not the book is currently borrowed.

Suppose in defining the subjectModified pointcut we made a mistake and left out, say, the second clause (concerning bkReturn()). What would happen? This is clearly a bug since the manager would then not have the correct information about the status of the book copy but there is nothing in the implementation in [7] to help prevent this. But our contract in Fig. 4, in particular the pre-condition of the subjectOther pointcut (line 32) will be violated when this method finishes (given the definition of Modified() in Fig. 5, line 44). Thus our approach enables us to specify the key properties expected of reusable aspects such as *Observer*.

4 Discussion

The main goal of our work was to develop techniques for specifying reusable abstract aspects so that when they are used in particular systems (by defining suitable concrete aspects), we are able to understand the behaviors of these systems by appealing to the specification of the reusable aspect and specializing it appropriately. We introduced the idea of defining a contract corresponding to the reusable aspect, with parts of the contract being expressed in terms of abstraction concepts that enable it to be both precise and be specialized in different ways to represent its use in different systems. One interesting and somewhat unexpected point that we encountered was that specifying the contract required us to define pointcuts/advice that were not required for the implementation of the aspect. Further, our contracts often allow us to identify ways to make abstract aspects more flexible than their original designs.

Our approach builds on existing work such as JML and Pipa but makes important extensions to them such as the introduction of the \oldM notation in specifying the behavior advice. One feature of JML and Pipa that we did not use in the current paper is the use of models in our specifications, choosing instead to refer directly to the internal components (such as c1.borrowed) of the objects. But it would be straightforward to allow for such models in our approach.

More challenging is the question of specification of *around* advice. The problem with such advice is that they effectively *replace* the method in question; or they may provide some code, followed by a call to the method (using the proceed mechanism) followed by additional code; they may even invoke the method multiple times! Developing ways to specify such advice is part of current work. Although [27]

deals with *around* advice, it does so by considering very restricted forms of such advice (some simple code, followed by a single proceed, followed by additional simple code). It would be interesting to consider more general forms of the advice.

References

[1] J. Aldrich. Open modules: Modular reasoning about advice. In *ECOOP*, 2005.

[2] E. Baniassad and S. Clarke. Theme: An approach for aspect-oriented analysis and design. In *ICSE*, 2004.

[3] L. Bass, M. Klein, and L. Northrop. Identifying aspects using architectural reasoning. In *Workshop on Early Aspects: AO Requirements Engineering and Architecture Design*, 2004.

[4] Y. Cheon and G. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *ECOOP*, 2002.

[5] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD*, 2007.

[6] S. Clarke and R. Walker. Composition patterns: An approach to designing reusable aspects. In *ICSE*, 2001.

[7] S. Clarke and R. Walker. Generic aspect-oriented design with theme/uml. In *Aspect-oriented software development*, pages 425–458. Addison-Wesley, 2005.

[8] P. Clemente, J. Hernández, J. Herrero, J. Murillo, and F. Sánchez. Aspect-orientation in the software lifecycle: Fact and fiction. In *Aspect-oriented software development*, pages 407–423. Addison-Wesley, 2005.

[9] C. Clifton and G. Leavens. Observers and assistants: A proposal for modular aspect-oriented reasoning. In *FOAL*, 2002.

[10] D. Dantas and D. Walker. Harmless advice. In *POPL*, 2006.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable OO Software*. Addison-Wesley, 1995.

[12] W. Griswold, K. Sullivan, Y. Song, M. Shonle, N. Tewari, Y. Cai, and H. Rajan. Modular software design with crosscutting interfaces. *IEEE Softw.*, 23(1):51–60, 2006.

[13] J. Grundy. Aspect-oriented requirements engineering for component-based software systems. In *RE*, 1999.

[14] J. Hannemann and G. Kiczales. Design pattern implementation in Java and AspectJ. In *OOPSLA*, 2002.

[15] I. Jacobson and P. Ng. *Aspect oriented software development with use cases*. Addison-Wesley, 2005.

[16] E. Katz and S. Katz. Incremental analysis of interference among aspects. In *FOAL*, 2008.

[17] R. Khatchadourian, J. Dovland, and N. Soundarajan. Enforcing behavioral constraints in evolving aspect-oriented programs. In *FOAL*, 2008.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *ECOOP*, 1997.

[19] G. Leavens, K. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in java. In *OOPSLA Companion*, 2000.

[20] T. Mikkonen. On objects, aspects, and specifications addressing their collaboration. In *Workshop on Early Aspects: AO Requirements Engineering and Architecture Design*, 2002.

[21] H. Ossher and P. Tarr. Hyper/j: multi-dimensional separation of concerns for java. In *ICSE*, 2001.

[22] A. Rashid, A. Moriera, and J. Araujo. Modularisation and composition of aspectual requirements. In *AOSD*, 2003.

[23] N. Soundarajan and J. Hallstrom. Responsibilities and rewards: Specifying design patterns. In *ICSE*, 2004.

[24] N. Soundarajan, J. Hallstrom, G. Shu, and A. Delibas. Patterns: from system design to software testing. *Innovations in Systems and Software Engineering*, 4(1):71–85, Apr. 2008.

[25] G. Sousa, S. Soares, P. Borba, and J. Castro. Separation of crosscutting concerns from requirements to design: Adapting the use case driven approach. In *Early Aspects Workshop at AOSD*, 2004.

[26] D. Wagelaar. A concept-based approach for early aspect modeling. In *Workshop on Early Aspects: AO Requirements Engineering and Architecture Design*, 2003.

[27] J. Zhao and M. Rinard. Pipa: Behavioral interface spec language for AspectJ. In *FASE*, 2003.