Improve Pointcut Definitions with Program Views

Zifu Yang, Tian Zhao
University of Wisconsin – Milwaukee, USA
{zifuyang,tzhao}@uwm.edu

ABSTRACT

Aspect-oriented programming languages select join points using pointcut constructs that depend on the syntactic structure of the base program. As the base program evolves, the pointcuts may no longer capture the intended set of join points. Also, pointcuts may select join points so that aspects can observe program behavior protected by encapsulation and this makes local reasoning difficult.

This work presents an approach for defining pointcuts based on program views, which are the abstractions of the classes and methods of the base program. Pointcuts are defined based on these views and syntactic changes in the base program will not affect the pointcuts if the base program is consistent with the views. A view also includes constraints to limit the set of join points that pointcuts can select and to help maintain modularity.

Categories and Subject Descriptors

D.2.2 [**SOFTWARE ENGINEERING**]: Design Tools and Techniques—*Modules and interfaces*; D.3.3 [**PROGRAMMING LANGUAGES**]: Language Constructs and Features

General Terms

LANGUAGES, DESIGN

1. INTRODUCTION

AspectJ inserts code of cross-cutting concerns into the base program at selected join points [8]. The pointcuts define sets of join points and they rely on the matching of types and method signatures. This can cause several problems. One is that the pointcuts are sensitive to the changes of the type names and method signatures. Any accidental matches or omissions of join points caused by the changes can result in unexpected behavior after aspect code is weaved into the base program. This problem, termed as *fragile pointcut problem* [12, 16, 7], hinders software evolution. Another problem is that aspects may break the encapsulation boundary of abstractions by selecting join points within the abstraction. As a result, common practices such as refactoring within an abstraction may render some of the existing aspects useless [19].

To prevent the first problem, the aspect programmers may need to update the existing aspects whenever the base program is changed or write more robust aspects to minimize the impact of the changes. The use of Java Annotations in pointcuts lessens the need of matching types by names and methods by signatures since base program can annotate types and methods with annotations and the join points are selected by the matching of the annotations. However, the aspect code is still directly dependent on the syntactic structure of base programs. The annotations may be placed incorrectly in the base program by programmers who are unaware of the existence of the aspects. Likewise, aspects using annotations may still detect and modify otherwise unobservable program behavior and break encapsulation.

In this paper, we present a method to address both problems using a join point model dependent on the *views* of the base program. A view includes abstractions for sets of types and methods to help creating robust pointcuts. This approach allows us to check for consistency of the pointcuts with the base program after each revision to avoid fragile pointcut problems. Also, programmers can place restrictions in the view to limit the sets of join points exposed to the aspects to improve modularity. Moreover, a view can relate to a particular cross-cutting concern of the base program and the pointcuts can be organized based on the views that they depend on.

The rest of the paper is organized as follows: Section 2 gives the motivation for program views. In Section 3, we use an example to illustrate the view constructs and view-based pointcuts. Section 4 explains the checking rules that we use to ensure that the base program and view-based pointcuts are consistent with the views. Section 5 outlines the implementation and Section 6 discusses the related work.

2. VIEW-BASED POINTCUTS

A view of the base program consists of modules and message interfaces. A module is a concept for a set of class types while a message interface is a concept for a set of methods. A module includes the properties: `sends` and `receives`, which specify the kinds of messages that a module can send and receive. If a module `C` sends message `M`, then the classes in `C` may call methods in `M`. If `C` receives message `M`, then the classes in `C` can receive calls to methods in `M`. However, the opposites may not be true.

A view-based pointcut descriptor (PCD) uses modules to specify the callers and receivers of calls and uses message interfaces to specify the invoked methods. Since modules may restrict the kind of message they can send or receive. view-based PCDs can only use valid combinations of modules and message interfaces. That is, if a view-based PCD captures calls of methods in `M` from module `C1` to module `C2`, then `C1` must send message `M` and `C2` must receive message `M`. For example, if a module `C` sends but not receives message

M, then the calls of methods in M between the classes in C cannot be captured by any view-based PCDs. So, if the classes of C form an abstraction, the above restriction may help protect its internal states from being exposed to the aspects.

We can specify the types in a module and the methods in a message interface using explicit enumeration, program annotations, or name patterns. If annotations are used, then the base program must be properly annotated. For a base program without annotation, we can first specify the modules and message interfaces using other approaches, create annotations for them, and then add annotations to the appropriate classes and methods of the base program. Also, after a base program is modified, it is checked to ensure that the definitions of the modules and message interfaces are still valid. The view-based PCDs are translated to the PCDs in AspectJ afterwards so that they are always consistent with the intention of the aspect programmers.

This paper will only use the pointcut designators within, call, and target of AspectJ to specify the calling context, called methods, and the receivers of join points. We can use other pointcut designators such as execute as well but it is not clear what kind of restrictions that we can impose on these pointcuts to improve modularity. To use designators such as withincode, handler, get, and set, we may need to design more constructs similar to the modules and message interfaces.

2.1 Motivation

The example in Figure 1 illustrates a program view.

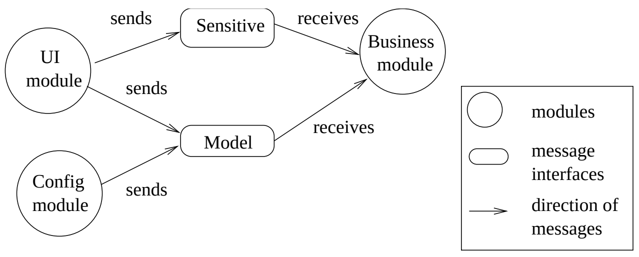

Figure 1: UI, Config, and Business are modules and Sensitive and Model are message interfaces. The arrow indicates the direction of sending and receiving messages for each module.

Here only the UI module sends Sensitive messages. This does not imply that classes in Config do not call methods in Sensitive message interface. It only says that in this particular view of the base program, calls of methods in Sensitive from Config are not considered.

Suppose that the classes in UI is annotated with @UI (the same for @Config), and Sensitive message is annotated with @Sensitive. After the base program is annotated, the PCD below can capture calls of Sensitive methods from UI classes.

```
pointcut sensitiveCalls() : within(@UI *)
                    && call(@Sensitive * * (..));
```

Not all PCDs based on a view are permitted. For example, since the Config module does not send Sensitive messages, the PCD below is not allowed.

```
pointcut badPointcut() : within(@Config *)
                    && call(@Sensitive * * (..));
```

This kind of restrictions may be useful for the following reason. The Sensitive message interface includes methods that require access control checks and UI module contains code that operates on behalf of untrusted users. So we can insert authorization code at the join points where the Sensitive methods are called. However, the Config module contains code that does not interact with untrusted users directly and its call to Sensitive methods do not require authorization. In fact, the Model and the Sensitive message interfaces may contain the same methods but since the PCDs are dependent on view concepts, there is no confusion. That is, the calls from Config to Business will not be selected as join points for the purpose of authorization.

A view is a kind of contract between a base program and pointcuts. Modules and message interfaces can specify the types and methods they represent using name patterns or enumeration. Since we insert the annotations generated from the view, programmers may update the base program using these annotations to change the types and methods included in a view. After a base program is revised, we can check whether the name patterns and annotations still match the same set of types and methods specified in the view. In any case, the differences of the view between revisions of base program are revealed to, instead of hidden from the aspect writers.

The changes in base program should also be consistent with the sends and receives properties of the modules. For example, if the classes of the UI module in Figure 1 no longer call methods in Model after a revision, the view should be updated so that UI does not send Model messages. View-based PCDs need not be aware of how the base program evolves if the constraints of the view remain the same.

A base program can have multiple views, each of which can be used by different sets of pointcuts. A view can be based on the architecture, a view can separate the functional code from the code for debugging and performance measurement, or a view can divide the code based the security properties. Pointcuts and even aspects can be organized by the different views that they depend on.

2.2 Representation of views

We encode views in Web Ontology Language (OWL) [15]. OWL is a language for representing domain-specific data models and can be used to reason about the objects in that domain and the relations between them. The main constructs of OWL include classes, individuals, properties, and (XML) data types. The individuals are instances of the OWL classes and properties describe the attributes of the individuals in terms of other individuals or data values.

We define OWL classes to represent Java types, methods, modules, message interfaces, and view-based PCDs. A base program's types, methods, view constructs, and their PCDs are individuals of these OWL classes. Also, we define OWL properties to relate these individuals.

3. VIEW CONSTRUCTS

This section explains the view constructs using an example (Figure 2) adapted from [9]. In the example, the line and point objects can be moved by calls to the moveBy() method or setX(), setY() method. The aspect UpdateSignaling inserts calls to update the display when the lines and points are moved.

The problem with UpdateSignaling is that its PCD change() may capture more join points than intended when the code of the Point class is modified. In particular, if the x, y fields of Point were made private, then the Line class must call the setter methods of these two fields to update its Point objects. In this case, the calls to Point.setX() and Point.setY() originated from Line.moveBy() are also captured by the PCD. This is not correct behavior since Display.update() is only supposed to be called when the Shape objects are actually moved. The additional calls to Display.update() caused by the aspect are not needed.

This problem can be corrected by changing advice to:

```
class ShapeClient {
  // calls methods of Point and Line class
}
interface Shape {
   public moveBy (int dx, int dy )
}
class Point implements Shape {
   int x, y
   public int getX() { return x; }
   public int getY() { return y; }

   public void setX(int x) { this.x = x; }

   public void setY(int y) { this.y = y; }

   public void moveBy(int dx, int dy){
      x += dx; y += dy;
   }
}
class Line implements Shape {
   private Point p1, p2;

   public Point getP1() { return p1; }
   public Point getP2() { return p2; }

   public void moveBy(int dx, int dy){
      p1.x += dx; p1.y += dy;
      p2.x += dx; p2.y += dy;
   }
}
aspect UpdateSignaling {
  pointcut change() :
          call(void Point.setX(int))
       || call(void Point.setY(int))
       || call(void Shape.moveBy(int,int));
  after() returning: change() {
     Display.update();
  }
}
```

Figure 2: The PCD change() would select unwanted join points if `Point.x` and `Point.y` were changed to private fields. The class `ShapeClient` calls `Shape.moveBy()`, `setX()`, and `setY()`.

```
after() returning:
        change() && !cflowbelow(change()) {
   Display.update()
}
```

The new advice only picks top-level calls to `Point.setX()`, and `Point.setY()`.

We deal with the above problem by defining a view with two modules: `Client` and `Host`, and one message interface `Move`. They can written in OWL as:

```
<Module rdf:ID="Client">
  <hasType rdf:resource="#ShapeClient" />
  <sends rdf:resource="#Move" />
</Module>

<Module rdf:ID="Host">
  <hasType rdf:resource="#Point" />
  <hasType rdf:resource="#Line" />
  <receives rdf:resource="#Move" />
</Module>

<Message rdf:ID="Move">
  <hasMethod rdf:resource="#setX" />
  <hasMethod rdf:resource="#setY" />
  <hasMethod rdf:resource="#moveBy" />
</Message>
```

Note that the tags `<Module>, </Module>` enclose the individual of the OWL class `Module`. The tag `<hasType ../>` is a property that specifies the Java types contained in a module and `hasMethod` property specifies the methods contained in a message interface. The string `rdf:ID="Client"` specifies the ID of an OWL individual as `Client.` This OWL individual can be refereed to elsewhere by `rdf:resource="#Client"`. Thus, the name `#Point` refers to the OWL individual for Java type `Point` as defined below

```
<JavaType rdf:ID="Point"> ... </JavaType>
```

OWL individuals for other types and methods such as `Line, setX`, `moveBy` can be defined similarly but they are not shown here.

The module `Host` receives but does not send `Move` message. A PCD based on the view above can only capture join points that are calls from `Client` to `Host` of the `Move` methods. Thus, the calls within `Host` module to the `Move` methods are hidden from aspects and they can change without affecting the aspects.

A view-based PCD called `change` can be written as:

```
<Pointcut rdf:ID="change">
  <within rdf:resource="#Client" />
  <call rdf:resource="#Move" />
  <target ref:resource="#Host" />
</Module>
```

The `within`, `call`, and `target` properties represent the corresponding pointcut designators in AspectJ. They are all optional and, for example, if the `target` property is missing, then all possible target modules, if any, will included when we translate a view-based PCD to AspectJ PCD.

The above view-based PCD can be translated to AspectJ as follows.

```
pointcut change() : within(ShapeClient)
   && (    call(Point.setX(int)
        || call(Point.setY(int)
        || call(Shape.moveBy(int, int)) )
   && ( target(Point) || target(Line) )
```

Here we mix the static `within` construct and dynamic `target` construct because we want view-based PCD to capture the sets of join points allowed by the view. For example, if there is a class `Circle` as defined below

```
class Circle implements Shape {
   ...
   public void moveBy(int dx, int dy) { ... }
}
```

then the view-based PCD will not capture the calls to the `moveBy` method on a `Circle` object since its type is not in the module `Host`.

Even if `Line.moveBy()` is revised to call `setX(), setY()` to change the x, y coordinates of points, there is no need to modify the view-based PCD since it does not capture the calls to `setX()`, `setY()` from types in module `Host`. With this approach, the PCDs are specific in the join points that they want to select. If aspect writers want to select join points that include calls to the setter methods from the `Line` class, they should change the `Host` module to

```
<Module rdf:ID="Host">
  ...
  <sends rdf:resource="#Move" />
</Module>
```

and then modify the view-based PCD:

```
<Pointcut rdf:ID="change">
  <within rdf:resource="#Client" />
  <within rdf:resource="#Host" />
  <call rdf:resource="#Move" />
  <target ref:resource="#Host" />
</Module>
```

3.1 Name patterns

Name patterns are convenient for matching methods or types with specific naming convention. For example, the PCD below can capture join points of calls to all methods with names starting with `set` so that if the base program is added a method `setXY(int, int)`, then calls to this method are also included.

```
pointcut setCall() : call(void *.set*(..));
```

However, this PCD can also select calls to a Point method `setColor()`, which may not be intended. Thus, we include an optional property `hasNamePattern` as shown below to specify the name pattern for types in a module and methods in a message interface. A method is included if its signature is matched by one of the patterns.

```
<Message rdf:ID="Move">
  <hasMethod rdf:resource="#setX" />
  <hasMethod rdf:resource="#setY" />
  <hasMethod rdf:resource="#moveBy" />

  <hasNamePattern rdf:datatype="xsd:string"/>
      Point.set*(..)
  </hasNamePattern>
  <hasNamePattern rdf:datatype="xsd:string"/>
      Shape.moveBy(..)
  </hasNamePattern>
</Message>
```

To avoid matching unintended methods or types, we compare the sets of methods and types matched with the patterns with the sets explicitly specified in the view and let the programmer to decide whether the matching is intended or accidental.

3.2 Annotations

If the base program contains annotations, we can specify types of a module and methods of a message interface with annotations as well. For example, we use the optional property `hasAnnotation` to associate the message `Move` with a OWL individual `MoveAnnotation` which corresponds to the Java annotation `@Move` (specified by the `hasName` property).

```
<Message rdf:ID="Move">
  ...

  <hasAnnotation rdf:resource="#MoveAnnotation" />
</Message>

<Annotation rdf:ID="MoveAnnotation">
  <hasName rdf:datatype="xsd:string/>
      @Move
  </hasName>
  <annotates rdf:resource="#setX" />
  <annotates rdf:resource="#setY" />
  <annotates rdf:resource="#moveBy" />
</Annotation>
```

Our implementation finds the methods annotated by `@Move` and set the `annotates` properties of `MoveAnnotation`. We also check whether these methods are the same as those explicitly listed in the message interface.

If the module `Client` and `Host` have annotations `@Client` and `@Host` respectively, then the view-based PCD `change` defined earlier can be translated to:

```
 pointcut change() :
             within(@Client *)
          && call(@Move * * (..))
          && @target(Host)
```

3.3 Expose context variable

The pointcuts defined so far do not expose context variables. If, for example, we want to identify the `Point` and `Line` objects that have been moved, then we define the pointcut `change()` as:

```
pointcut change(Shape s) :
 ...
 && target(s)
```

However, the view of the base program has no information about the type of the classes in the `Host` module. It is likely that not all classes included in the `Host` module have the target type specified in the PCD. For example, if the above PCD is changed to

```
pointcut change(Line s) :
 ...
 && target(s)
```

then the join point selected will only be the calls to the `Line` objects. To prevent this problem, we can include an optional property `hasSuperType` to specify the super type of the classes in a module. For example,

```
<Module rdf:ID="Host">
  ...

  <hasSuperType rdf:resource="#Shape" />
</Module>
```

If a pointcut exposes context variable involving `Host`, it can declare the variable's type to be `Shape`. Similarly, if the methods in a message interface have parameters at the same position with the same type, we can specify that in the message interface so that argument variable at the position can be selected in the pointcuts.

4. CHECKING BASE CODE AND VIEWS

In this section, we summarize the definitions of view constructs, view-based PCDs, and explain the consistency checks of base program against views.

A view includes a set of modules and message interfaces. All modules in a view contain disjoint sets of types. We use the following symbols to represent the IDs of OWL individuals: `C` for modules, `M` for message interfaces, `T` for Java types, `A` for Java annotations, `P` for method parameters, `PC` for pointcuts, and `Mth` for methods.

A module may have the following form.

```
<Module rdf:ID="C"
  <sends        rdf:resource="#M1" />
  <receives     rdf:resource="#M2" />
  <hasType      rdf:resource="#T1" />
  <hasSuperType rdf:resource="#T" />
  <hasPattern   rdf:type="xsd:string">
     typePattern
  </hasPattern>
  <hasAnnotation rdf:resource="#A" />
</Module>
```

All properties of Module are optional. If a module `C` is defined as above, then the types in `C` must call some methods in `M1` and implements some methods in `M2`. Also, the types in `C` have super type specified by `T`. Moreover, the type specified by `T1` matches the pattern `typePattern` and it has the annotation specified by `A`.

A message interface may have the following form. All its properties are optional.

```
<Message rdf:ID="M"
  <hasMethod            rdf:resource="#Mth" />
  <hasCommonParameter   rdf:resource="#P" />
  <hasPattern rdf:type="xsd:string">
```

```
     methodPattern
   </hasPattern>
   <hasAnnotation        rdf:resource="#A" />
</Message>
```

The property `hasCommonParameter` specifies the parameter of all the methods in `M` that has the same type and position. The parameter `P` is defined as

```
<Parameter rdf:ID="P"
   <hasType rdf:resource="#T" />
   <hasPosition rdf:type="xsd:integer" />
     1
   </hasPosition>
</Parameter>
```

If a message interface `M` is defined as above, then the method specified by `Mth` must match pattern `methodPattern` and has the annotation specified by `A`. Also, the method must have a parameter with type `T` and is the first parameter of the method.

A view-based PCD may have the following form.

```
<Pointcut rdf:ID="PC"
   <within       rdf:resource="#C1" />
   <call         rdf:resource="#M" />
   <target       rdf:resource="#C2" />
   <thisType     rdf:resource="#T1" />
   <targetType   rdf:resource="#T2" />
   <args         rdf:resource="#P" />
</Parameter>
```

Again, all properties are optional. If a pointcut `PC` is defined as above, then the module `C1` must send message `M` and `C2` receives message `M`. Also, the properties `thisType` and `targetType` are used to expose the context variables of the executing context object and the receiver object. The type `T1` must be the super type of the types in module `C1` and `T2` must be the super type of the types in `C2`. The property `args` specifies the parameter variable that can be exposed at the join points. The parameter specified by `P` must be a common parameter of the methods in `M`.

5. IMPLEMENTATION

We implement the editor and checker of program views as a Plugin to the Protege OWL development environment [11]. The structure of the implementation is illustrated in Figure 3.

The editor uses a predefined OWL file with classes and properties for the Java types, methods, annotations, parameters, modules, message interfaces, and view-based PCDs. We load the base program as jar files into the editor and populate the individuals of OWL classes for Java types and methods.

With the editor, an user can choose any of following three methods to define modules and message interfaces.

1. Enumerate the Java types of a module and methods of a message interface.

2. Use type and method name patterns to specify the types of a module and methods of a message interface.

3. If the base program is annotated, use annotations to specify the types of a module and methods of a message interface

If we have defined the types of a module using first two methods, we can still create a module annotation and then add it to the corresponding class types in the base program. After modules and message interfaces are defined, we check their consistency programmatically using the rules explained in Section 4. Then, we validate the view-based PCDs and translate them to AspectJ PCDs.

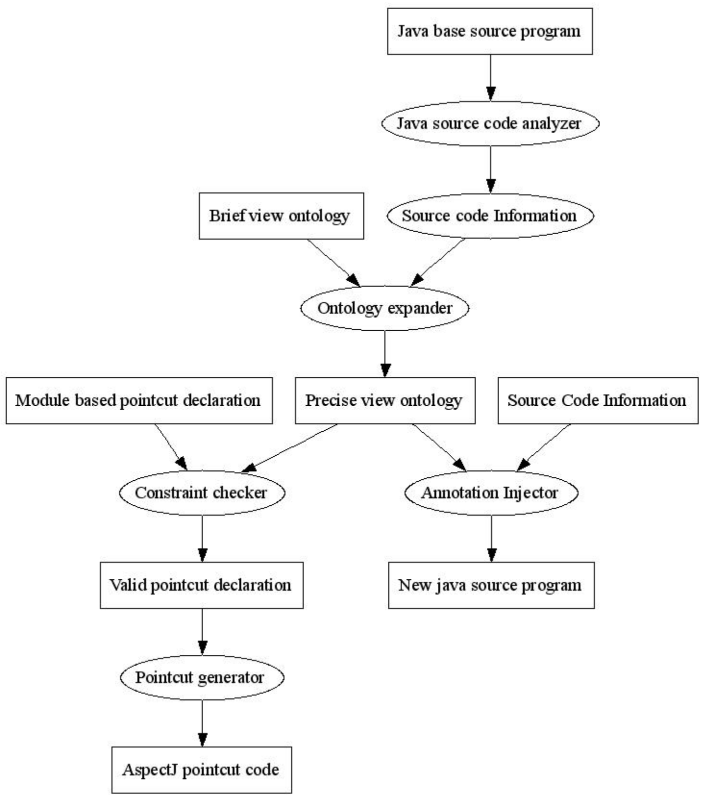

Figure 3: Illustration of the implementation.

6. RELATED WORK

The model-based pointcuts [7] of Kellens et al. allow pointcuts be based on conceptual models instead of the base program. This indirection prevents accidental matches or omissions of join points because the conceptual model is synchronized with the base program as it evolves and constraints are in place to check if the concepts of the model still match source entities in the base program. They introduced an instantiation of the model-based pointcuts using the formalism of intensional views [14] to express view-based pointcuts. The intensional view model is synchronized with the base program using constraints including alternative intensions and intensional relations. This work can be seen as another instantiation of their conceptual model though we only support a few pointcut constructs. Our concepts are modules and message interfaces and we check consistencies of the base programs and the concepts based on the alternative definitions of these concepts. What is different from intensional views is that our program views contain restrictions on how the modules and message interfaces may be used in combination in a view-based PCD. Any aspects using our view-based PCDs will follow these restrictions. This is intended to improve the modularity of AOP programs.

PCDiff [12, 16] is a tool to compare the sets of join points matched by pointcuts on different versions of the base program. It helps aspect programmers to identify accidental matches or omissions of join points by the pointcuts after program changes. PCDiff is not as effective if the base program adds new concepts that should be included in the captured join points.

Havinga et al. [6] introduce a way to superimpose annotations to base programs using pointcut constructs. Other pointcuts can depend on the superimposed annotations so that the annotations are not scattered in the base program and they can be application specific. Our method is not as flexible since we can only insert annotations to base program based on the defined modules and message interfaces. Kiczales and Mezini [10] discussed the interaction of annotation, advice, pointcut, and procedure for the purpose of separation of concerns. They also explained the trade-off between the pointcuts based on annotations, enumerations, and name-patterns. We can specify the concepts used in view-based PCDs with all three approaches. The generated AspectJ pointcuts could be annotation-based, enumeration-based, or name-pattern based. Each style of pointcuts is a concrete representation of the view-based pointcuts and our consistency checking ensures that they are equivalent.

Open modules [1] and aspect-aware interfaces [9] address the modularity problem of aspects by relating the interfaces of base program to the advices. Kiczales and Mezini [9] proposed the concept of aspect-aware interfaces so that aspect-oriented programming can be used to enable modular reasoning in the presence of cross-cutting concerns. Aspect-aware interfaces list the methods affected by the advices. In contrast, program views are not as detailed, and types and methods of base program are related to the PCDs indirectly through views. Open modules [1] export functions and pointcuts so that clients of the modules can place advice on exported pointcuts and on external calls to the exported functions. This can protect the internal states of the modules from aspects. Open modules provide one kind of abstraction for base program, while program views can partition base code into multiple sets of modules. Program views also restrict the calls between modules that can be matched by pointcuts. Join point encapsulation [13] uses a specialized advice construct to restrict the sets of join points exposed to the aspects. This can also improve modularity and make it easier to reason about the program behavior as base program evolves.

Crosscut Programming Interface (XPI) [4] decouples base code from aspect code through design rules [17], which require the base code to satisfy some preconditions at the join points exposed by pointcuts and require the aspect code to satisfy some postconditions after its execution. Base programs and aspects can evolve independently if they follow the design rules. In program views, the `sends` and `receives` properties of modules are a weak form of pre- and postconditions for base programs and aspects. For example, the base program should ensure that if a module `C` sends message `M`, then the classes of `C` calls methods in `M` and the aspects can only advise join points of calls to methods in `M` from the classes in `C`.

Altman et.al. [2] use ontologies [5] to model the architecture of applications. They insert annotations based on the concepts defined in the ontology into the base programs, and write aspects using a domain-specific language based on such ontology. Their join point model mainly includes the keywords: sender, receiver, and message, which specify the caller, callee, and the invoked method by matching the concepts in the ontology. Their aspect language is limited to these constructs and does not expose context variables such as the target and the arguments of a call.

The concepts of program views are relatively simple compared to other models such as Conceptual Modules [3], which are used to analyze the logical structure of C code iteratively with tool support. Multi-dimensional separation of concerns of Tarr et al. [18] presents more complex scenario of how concerns can be specified separately as hyperslices and composed together as hypermodules using central rules. Aspects can be seen as hyperslices but each aspect has its own rules (pointcuts) to specify how it is woven in the base program. Since program view can restrict the join points captured by view-based PCDs, it may serve as a form of central rules for weaving related aspects into the base classes.

7. CONCLUSION

Aspect-oriented languages allow us to modularize cross-cutting concerns but the interaction of aspects with the base program can be hard to comprehend. Maintaining the correctness of pointcuts as base program evolves can also be difficult because the pointcuts directly depends on the structure of the base program.

We have presented an approach to write more robust pointcuts based on program views. A program can have several domain-specific views and pointcuts dependent on the views are not directly affected by the evolution of the base program. We check the base program after its revision to ensure that it is consistent with the views and we check the pointcut definitions to make sure that they satisfy the constraints of the views. The reader of a program can use program view as a guide to understand the interaction of the aspects with the base program.

For future work, we plan to experiment on an open source project to evaluate the usefulness of program views. We also will examine the interaction of other pointcut designators such as cflow and execution with program views.

8. REFERENCES

[1] J. Aldrich. Open Modules: Modular Reasoning about Advice. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 144–168, 2005.

[2] R. Altman, A. Cyment, and N. Kicillof. On the Need for SetPoints. In *European Interactive Workshop on Aspects in Software*, 2005.

[3] E. L. A. Baniassad and G. C. Murphy. Conceptual Module Querying for Software Reengineering. In *Proceedings of International Conference on Software Engineering*, 1998.

[4] W. Griswold, M. Shonle, K. Sullivan, Y. Song, Y. Cai, N. Tewari, and H. Rajan. Modular Software Design with Crosscutting Interfaces. *IEEE Software*, pages 51–60, January/February 2006.

[5] T. Gruber. Toward Principles for the Design of Ontologies Used for Knowledge Sharing. In *International Workshop on Formal Ontology*, 1993.

[6] W. Havinga, I. Nagy, and L. Bergmans. Introduction and Derivation of Annotations in AOP: Applying Expressive Pointcut Language to Introductions. In *European Interactive Workshop on Aspects in Software*, 2005.

[7] A. Kellens, K. Mens, J. Brichau, and K. Gybels. A Model-driven Pointcut Language for More Robust Pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 501–525, 2006.

[8] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. V. Lopes, J. M. Loingtier, and J. Irwin. Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 220–242, 1997.

[9] G. Kiczales and M. Mezini. Aspect-Oriented Programming and Modular Reasoning. In *Proceedings of International Conference on Software Engineering*, 2005.

[10] G. Kiczales and M. Mezini. Separation of Concerns with Procedures, Annotations, Advice and Pointcuts. In *Proceedings of the European Conference on Object-Oriented Programming*, 2005.

[11] H. Knublauch, R. W. Fergerson, N. F. Noy, and M. A. Musen. The Protege OWL Plugin: An Open Development Environment for Semantic Web Applications. In *Third International Semantic Web Conference*, 2004.

[12] C. Koppen and M. Stoerzer. PCDiff: Attacking the Fragile Pointcut Problem. In *First European Interactive Workshop*

on Aspects in Software, 2004.

[13] D. Larochelle, K. Scheidt, and K. Sullivan. Join Point Encapsulation. In *Proceedings of the Workshop on Software Engineering Properties of Languages for Aspect Technologies*, 2003.

[14] K. Mens, A. Kellens, F. Pluquet, and R. Wuyts. Co-evolving code and design with intensional views - a case study. *Computer Languages, Systems and Structures*, 32(2-3):140–156, 2006.

[15] M. K. Smith, C. Welty, and D. L. McGuinness. Web Ontology Language Guide. http://www.w3.org/TR/owl-guide/, 2004.

[16] M. Stoerzer and J. Graf. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 653–656, 2005.

[17] K. Sullivan, W. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information Hiding Interfaces for Aspect-Oriented Design. In *Proceedings of the Symposium on the Foundations of Software Engineering joint with the European Software Engineering Conference (ESEC/FSE)*, pages 166–175, 2005.

[18] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N Degrees of Separation: Multi-Dimensional Separation of Concerns. In *Proceedings of International Conference on Software Engineering*, 1999.

[19] M. Wand. Understanding Aspects. In *Proceedings of the eighth ACM International Conference on Functional Programming*, pages 299–300, 2003.