Instituto de Matemática e Estatística da Universidade de São Paulo

Pointcuts abertos

Monografia para a disciplina MAC5701 – Tópicos em Ciência da Computação

Cristiano Malanga Breuel
Orientador: Prof. Francisco Reverbel
São Paulo, junho de 2005.

Índice

Introdução

Este estudo pretende reunir e analisar os trabalhos que têm sido apresentados no sentido do desenvolvimento de *pointcuts* abertos, na área de Programação Orientada a Aspectos. Por *pointcuts* abertos, entendam-se formas de definição de pointcuts que permitam uma flexibilidade maior em relação às linguagens existentes. A flexibilidade, aqui, pode ser medida em duas dimensões: a granularidade e profundidade da informação a que se tem acesso na definição de um *pointcut*, e a capacidade de combinação dessa informação.

Nesta seção introduzimos os conceitos fundamentais da área de Programação Orientada a Aspectos, sobre a qual desenvolveremos o trabalho.

1.1. AOP

A programação orientada a aspectos (POA, ou AOP como é conhecida em inglês), é um paradigma de programação que surgiu com a intenção de modularizar características de um programa cuja implementação não pode ser claramente modularizada nos paradigmas tradicionais. Essas características são chamadas de *crosscutting concerns*, ou "interesses transversais", e sua implementação através de técnicas como a POO (Programação Orientada a Objetos) geram código não localizado, entrelaçado (*tangled*) ao restante do sistema.

A POA não pretende substituir a POO, mas complementá-la. Por isso, todas as implementações de linguagens AOP existentes recaem sobre duas formas de implementação: ou são extensões de linguagens existentes, ou são arcabouços escritos sobre uma linguagem existente. Em ambos os casos, há uma etapa em que o código orientado a aspectos é inserido em pontos específicos do código base, num processo chamado de *aspect weaving* (tecelagem de aspectos, combinação de aspectos).

Para atingir o isolamento de interesses transversais, a POA introduz um novo tipo de abstração: o *aspecto*. Um aspecto consiste em partes que implementam o interesse e partes que definem onde essa implementação deve ser inserida no programa-base. A implementação dos interesses é composta por *advices* (conselhos, adendos) e *inter-type declarations* (declarações intertipos), e é feita na mesma linguagem em que se baseia a linguagem ou arcabouço orientado a aspectos.

As partes de um aspecto que definem onde ele deve se aplicar são chamadas de *pointcuts* ("corte pontual", "conjunto de pontos de junção" são algumas das traduções propostas), ou *crosscutting parts* (esta uma terminologia menos utilizada). Na seção seguinte, examinaremos este conceito em mais detalhes.

1.2. Joinpoints e Pointcuts

Os *joinpoints* (pontos de junção) são os "ganchos" de um programa nos quais os aspectos podem ser "encaixados", ou seja, onde o código dos conselhos é inserido como parte do programa. Os tipos de pontos de junção mais comuns são chamadas e execuções de métodos, leitura e escrita de campos, tratamento de exceções e inicialização de classes.

Os *pointcuts* são expressões definidas pelo programador que, aplicadas sobre o conjunto de pontos de junção de um programa, selecionam um conjunto deles. O conjunto gerado pelo *pointcut* servirá de base para a aplicação de aspectos.

Nas linguagens atuais, os *pointcuts* dividem-se em dois tipos: os primitivos e os definidos pelo usuário. Os primeiros são pré-definidos como palavras-chave na linguagem, os segundos são compostos pelo programador através da combinação lógica dos primeiros.

Na tabela abaixo [12], vemos exemplos de modelos de *pointcuts* das linguagens disponíveis atualmente:

	AspectJ	**AspectWerkz**	**JBoss AOP**	**Spring AOP**
invocation	{method, constructor, advice} x {call, execution}			method execution
initialization	instance, static, pre-init	instance, static	instance	-
access	field get/set			-
exception handling	handler		(via advice)	
control flow	cflow, cflowbelow		(via specified call stack)	cflow
containment	within, winthincode	within, winthincode, has method/field	within, winthincode, has method/field, all	-
conditional	if	-	(via Dynamic cflow)	custom pointcut
pointcut matching	signature, type pattern, subtypes, wild card, annotation		signature, instanceof, wild card, annotation	regular expression
pointcut composition	&&, ||, !			&&, ||
extensibility	abstract pointcuts	overriding, advice bindings		

Em geral, as linguagens de pointcuts permitem:

- Capturar chamadas e execuções de métodos, leitura/escrita de campos, tratamento de exceções e inicialização de classes;
- Restringir a captura por fluxo de controle (exemplo: chamadas ao método m1 efetuadas no fluxo de controle do método m2), tipos onde os eventos ocorrem (exemplo: chamadas a m1 efetuadas pela classe c1) e métodos com parâmetros de tipos específicos (exemplo: chamadas a m1 que recebam uma instância da classe c2 como primeiro parâmetro);
- Combinar esses predicados através de lógica booleana.

Como vemos, as diversas linguagens orientadas a aspectos em uso atualmente têm características muito semelhantes no que refere à definição de *pointcuts*. Por isso, no restante da discussão, compararemos as novas propostas às "linguagens em uso atual", genericamente, pois todas têm expressividade equivalente. Nos casos em que são feitas comparações explícitas com alguma linguagem específica, como o AspectJ, a comparação pode ser generalizada para todas as linguagens mencionadas nesta seção.

1.3. Qualidade de um pointcut

Quando um programador define um *pointcut*, ele tem em mente um conjunto de pontos de junção que devem fazer parte dele. Esse conjunto normalmente é definido por características em comum entre os pontos de junção, que fazem com que seja necessária a aplicação de um aspecto a eles. Em uma situação ideal, a definição do *pointcut* seria a própria definição semântica dos critérios idealizados pelo programador.

Definimos a qualidade da definição de um *pointcut* como a sua capacidade de atender aos seguintes requisitos:

- **Resistência a mudanças**: mudanças no programa devem afetar o mínimo possível o conjunto gerado pelo *pointcut*, e consequentemente evitar a necessidade de mudanças em sua definição. Em particular, quando novos elementos são adicionados ao programa, o aspecto deve ser automaticamente aplicado onde for necessário, e apenas nesses pontos.
- **Clareza de intenção**: a definição de um *pointcut* deve transmitir, tanto quanto possível, e de forma simples e clara, a intenção de quem o definiu. Ou seja, ao examinar tal definição, um programador sem conhecimento prévio deve ser capaz de entender os critérios que o criador do *pointcut* tinha em mente ao criá-lo.

Utilizaremos esse conceito de qualidade do *pointcut* e suas variáveis para comparar as linguagens orientadas a aspectos atuais com diversas propostas de novas linguagens ou melhorias das existentes, que estão sendo pesquisadas. Como veremos, todos esses trabalhos buscam melhorar a qualidade da definição dos pointcuts, seja em relação à clareza da definição e/ou à resistência a mudanças.

1.4. Estilos de definição de pointcuts

Durante o desenvolvimento da programação orientada a aspectos e de programas escritos nela, observamos que emergiram algumas formas básicas de especificar *pointcuts*. A seguir, classificamos estas formas de declaração, analisando suas vantagens e desvantagens.

1.4.1. Enumeração

A forma mais básica de definir um *pointcut* é enumerando-se os pontos de junção que devem compô-lo. Esta enumeração pode ser um a um ou através de alguma convenção de nomenclatura. Os *pointcuts* abaixo ilustram essa forma:

```
pointcut DBActivity() :
 call(void Person.updateName(String)) ||
 call(void Person.updateAge(int)) ||
 call(void Company.insertPerson(Person))
||
 call(void Company.deletePerson(Person));
```

```
pointcut DBActivity() :
 call(void *.update*(..)) ||
 call(void *.insert*(..)) ||
 call(void *.delete*(..));
```

Podemos usar como metáfora para esta forma de definição um “recrutamento”. Neste caso, o recrutador é o programador do aspecto (ou da integração do aspecto a um programa específico), que escolhe os pontos de junção que deseja incluir no *pointcut*.

A principal desvantagem desta forma de declaração de *pointcuts* é que inclusões de pontos de junção não vão ser automaticamente consideradas, a menos que se encaixem na convenção de nomenclatura (no caso do segundo *pointcut*). Por outro lado, a definição é precisa (especialmente no caso do primeiro exemplo), evitando que sejam incluídos inadvertidamente pontos de junção indesejáveis.

1.4.2. Anotações

Uma outra forma de determinar os pontos de junção de um *pointcut* é marcá-los com alguma forma de meta-informação, o que pode ser feito através de "anotações" de métodos na linguagem Java. Neste caso, o programador dos métodos é quem decide que o método tem uma determinada propriedade e marca-o dessa forma, permitindo que o *pointcut* o selecione. Vemos um exemplo disso no trecho abaixo:

```
public class Person {

  @DBActivity
  public void updateName(String newName) {
    DBManager.openConnection();
    ...
  }

}

aspect DBConnectionDisposal {

  pointcut DBActivity() :
    call(@DBActivity * *(..));

  after() : DBActivity() {
    DBManager.closeConnection();
  }

}
```

Seguindo com as metáforas, podemos dizer que o método em questão "alista-se" para fazer parte do *pointcut*, ao declarar-se possuidor de uma propriedade interessante para este. Em comparação com a primeira abordagem, esta é mais precisa. Não é mais necessário confiar em convenções de nomenclatura, que são mais frágeis que anotações (as anotações têm tipos definidos estaticamente). Além disso, elimina-se a possibilidade de inclusão involuntária de pontos de junção em um pointcut, pois o programador deve explicitar a propriedade de seu método.

No lado negativo, ainda é necessário depender da aderência do programador a uma convenção. Além disso, esta técnica é mais intrusiva, pois pode exigir mudanças no programa-base para acomodar novos aspectos, reduzindo o isolamento.

1.4.3. Declarações semânticas

Finalmente, os *pointcuts* podem ser declarados baseando-se na semântica da situação que se deseja capturar. O trecho abaixo exemplifica este estilo:

```
public class Person {

  public void updateName(String newName) {
    DBManager.openConnection();
    ...
  }

}

aspect DBConnectionDisposal {

  pointcut* DBActivity(): pcflow(execution(void Server.initiateProcess()))
    && call(* DBManager.openConnection());
```

```
  after call(<DBActivity()>)(): {
    DBManager.closeConnection();
  }

}
```

Neste caso, a idéia do programador era capturar todas as execuções do método `acceptConnection()` que tenham aberto a conexão com o banco de dados, para garantir que ao final esta conexão será fechada.

O pseudo-*pointcut* DBActivity é definido com base no pseudo-*pointcut* `pcflow`, proposto por Kiczales [10], que indica uma previsão de fluxo de controle. Estes não são *pointcuts* reais porque eles apenas geram uma lista de pontos de junção, que pode ser usada como argumento na composição de um *pointcut* real. No exemplo, o *pointcut* real é "`call(<DBActivity()>)`", que denota o conjunto de chamadas selecionadas por `DBActivity`.

Este *pointcut* é muito mais resistente a mudanças que os anteriores, pois não se baseia em convenções de nomenclatura nem em declarações de meta-informação. Não importando como o programa foi implementado, se uma conexão ao banco de dados foi aberta, ela será fechada, garantindo que a intenção do programador do aspecto foi realizada.

Comparado às metáforas dos *pointcuts* anteriores, neste caso o que teríamos seria uma espécie de "filtragem", em que os pontos de junção selecionados seriam os que efetivamente possuem a propriedade de abrir uma conexão com o banco de dados, não importa quais sejam os seus nomes ou propriedades.

Como podemos ver, as declarações semânticas são o ideal para a programação orientada a aspectos. No entanto, as capacidades das linguagens orientadas a aspectos atuais nem sempre permitem a criação de *pointcuts* deste tipo. O exemplo acima está em pseudo-código, pois o *pointcut* `pcflow` não existe em nenhuma linguagem comercialmente disponível.

2. Limitações das linguagens em uso atual

Os recursos para definição de *pointcuts* à disposição nas linguagens atuais atendem à maioria das necessidades. Mas, como veremos na seção 2.1, muitas vezes eles não permitem criar *pointcuts* de boa qualidade, tanto no que se refere à resistência a mudanças quanto à clareza.

A principal limitação dessas linguagens é o fato de não serem turing-completas. No caso do arcabouço Spring-AOP, é possível utilizar Java para definir *pointcuts* "personalizados". A API à disposição para isso, no entanto, é a interface de reflexão da própria linguagem, que não oferece granularidade suficiente para examinar um ponto de junção internamente (por exemplo, que acessos a campos podem existir dentro da execução de um método).

A seção 2.1 evidencia essas deficiências.

2.1. Exemplos motivadores

Apresentamos nesta seção alguns exemplos de problemas que evidenciam a necessidade de melhoramentos ao modelo de *pointcuts* das linguagens orientadas a aspectos. Apresentamos também uma solução possível para cada um desses problemas na linguagem orientada a aspectos mais popular, o AspectJ. Nas seções dedicadas a cada uma das propostas pesquisadas, é apresentada a solução dada por elas para alguns desses problemas.

2.1.1. Editor de figuras

O exemplo mais utilizado em tutoriais e apresentações introdutórias sobre programação orientada a aspectos é o de uma hipotética aplicação gráfica [2]. Nesta aplicação, existem diversos elementos gráficos, como linhas e retângulos, representados por classes. Quando é invocado um método que altera o estado de uma instância de alguma dessas classes, o método `redraw()` deve ser invocado para atualizar a exibição do elemento. Na Figura 1, vemos o diagrama UML desse editor de figuras.

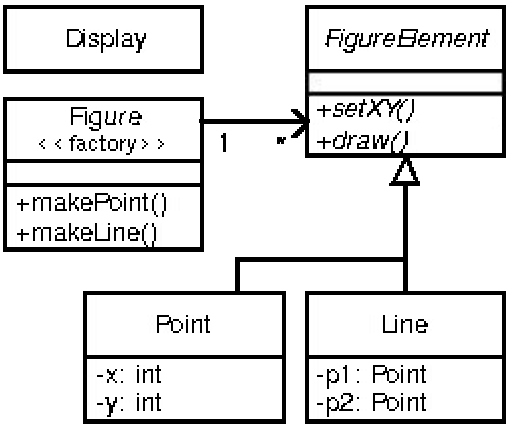

Figura 1 Diagrama UML do editor de figuras

Embora seja um exemplo simples e sirva de exemplo de aplicação de programação orientada a aspectos, este é também um bom exemplo das limitações das formas atuais de especificação de *pointcuts*. As listagens abaixo mostram duas possíveis soluções para o problema, em AspectJ.

```
aspect DisplayUpdating {
 pointcut move() :
  call(void Line.setP1(Point)) ||
  call(void Line.setP2(Point)) ||
  call(void Point.setX(int)) ||
  call(void Point.setY(int));

 after() returning :
   move() && target(fe) {
  fe.redraw();
 }
}
```

```
aspect DisplayUpdating {
 pointcut move() :
  call(void FigureElement+.set*(..));

 after() returning :
   move() && target(fe) {
  fe.redraw();
 }
}
```

Na primeira solução, todos os métodos que afetam as figuras são enumerados no *pointcut*. Se um programador adicionar um novo método que altera o estado de uma figura e não alterar o *pointcut*, este deixará de ser completamente efetivo. O mesmo ocorre se for criada uma nova subclasse de `FigureElement`.

A segunda solução é mais genérica, pois inclui qualquer subtipo de `FigureElement`, e seleciona todos os métodos cujo nome se inicia por "set". Neste caso, a adição de um novo método já não quebra o *pointcut*, desde que o programador siga a convenção de iniciar o método por "set". Se o método, no entanto, for chamado de `change()`, ele também não será incluído no *pointcut*.

Em nenhum dos casos, o *pointcut* transmite ou implementa diretamente a intenção do programador, que era capturar todas as mudanças de estado das figuras.

Este problema, embora pareça específico, na verdade representa uma classe ampla de situações às quais a programação orientada a aspectos pode ser aplicada. Genericamente, podemos descrever essa classe como aquela dos problemas em que é necessário tomar ou não uma ação com base nas ações tomada internamente dentro de certo fluxo de controle.

Isso pode incluir a persistência de um objeto se o seu estado foi alterado, o fechamento de uma conexão de rede caso ela tenha sido aberta, ou a negação de acesso a um método se ele poderia fazer uma alteração não autorizada no sistema. Este caso difere um pouco pelo fato de se basear em atos no futuro, ou seja, o conselho tem que agir antes que as ações em questão efetivamente ocorram. O item a seguir ilustra essa aplicação.

2.1.2. Segurança

O exemplo abaixo ilustra um aspecto de controle de acesso a uma lista de compras, extraído (com modificações) de [14]. A intenção do programador é impedir que um cliente altere a cesta de outro.

```
public class ShoppingCart {

  private Set items;
  private double total;
  public Customer receiver;

  ShoppingCart(Customer receiver) {
    this.receiver = receiver;
    items = new HashSet();
    total = 0.0;
  }

  public void addItem(Integer itemNr) {
```

```
    items.add(itemNr);
    total += Database.loadPrice(itemNr);
  }

  public void removeItem(Integer itemNr) {
    items.remove(itemNr);
    total -= Database.loadPrice(itemNr);
  }

}

public abstract aspect ItemChanges {

  pointcut itemChanges(Customer c, ShoppingCart s) :
    this(c) && target(s) && call(* ShoppingCart.*Item(..));

}

public aspect Authorization extends ItemChanges {

  before (Customer c, ShoppingCart s) : itemChanges(c, s) {
    if (!mayAccess(c, s)) {
      throw new AccessException("Illegal Access - denied.");
    }
  }

  private boolean mayAccess(Customer c, ShoppingCart s) {
    return c.equals(s.receiver);
  }

}
```

Suponha-se agora que foi adicionado um novo recurso à aplicação de compras, que permite a um cliente compartilhar a sua cesta de compras, como uma “wishlist”. Esta funcionalidade é implementada por um novo método na classe `ShoppingCart`, chamado `showItem()`. Esta nova funcionalidade não vai funcionar corretamente, pois o aspecto de autorização vai impedir o acesso do amigo à cesta, devido ao nome do método.

Para resolver o problema, seria necessário alterar o *pointcut*, passando a enumerar todos os métodos que alteram a cesta. Nesse caso, de fosse criado um novo método de alteração da cesta e ele não estivesse listado no *pointcut*, a cesta estaria desprotegida. No caso do *pointcut* com definição via *wildcard*, esse risco também existe, mas é um pouco menor.

3. Abordagens

Todas as abordagens existentes no sentido de linguagens de *pointcuts* mais flexíveis têm em comum o fato de transformarem as definições de *pointcuts* em meta-programas, atuando sobre um modelo do programa sobre o qual o aspecto deve ser aplicado.

Uma classificação imediata que pode ser feita entre essas propostas é pelo paradigma da linguagem usada como base. Esses paradigmas correspondem aos principais existentes na programação de forma geral. Por isso, agrupamos os trabalhos desta forma.

3.1. Programação Funcional

3.1.1. XQuery/BAT

A proposta do trabalho [6] é expressar definições de *pointcuts* como consultas na linguagem XQuery. O XQuery é uma linguagem funcional, criada para permitir consultas em um meta-modelo da linguagem de marcação XML. Para permitir o uso do XQuery, é utilizado um arcabouço que converte *bytecodes* Java em uma representação XML. Assim, os *pointcuts* podem ser definidos como consultas em XQuery sobre esse meta-modelo do programa.

Segundo os autores, a programação funcional permite que as declarações de *pointcuts* sejam curtas, precisas e declarativas. Em contraste, implementações de *pointcuts* em linguagens como AspectJ são complicadas, e por isso é necessário existir uma especificação formal da sua semântica.

Outra vantagem desse tipo de declaração seria a facilidade de composição de *pointcuts*. Ao contrário da semântica limitada do AspectJ, o uso de uma linguagem *turing-completa* como o XQuery permite criar novos *pointcuts* usando o resultado de outros *pointcuts* na composição.

Do ponto de vista da implementação, a imagem do programa sobre a qual as consultas são executadas é uma tradução para XML dos *bytecodes* do programa, gerada pelo arcabouço BAT, criado pela equipe dos próprios autores. Esta transformação gera uma imagem estática e de baixo nível do programa.

Como exemplo da expressividade dessa abordagem, os autores dão a seguinte solução para o problema do item 2.1.1:

```
1.  declare function pcflow($all as element()*, $m as element()*) as element()* {
2.    let $pcfl1 := $all//method[@name = $m//invoke/@method] except $m//method
3.    return if (empty($pcfl1)) then $m else pcflow($all, $m union $pcfl1)
4.  }

5.  $all//put[@name = pcflow($all,
6.      subtypes($all,$all/class[@name="FigureElement"])
7.          /method[@name="draw"])//get/@name ]
```

O conteúdo das linhas 5 a 7 seria o *pointcut* ao qual seria aplicado o conselho de atualização da exibição das figuras.

Embora esta solução seja melhor do que as possíveis com o AspectJ, ela ainda não é ideal. Isto porque a predição do fluxo de controle é feita baseada apenas em dados de uma representação estática do programa. Os autores reconhecem isso, e tratam a análise de informações dinâmicas como trabalho futuro.

3.2. Programação Lógica

Diversos trabalhos foram feitos no sentido de utilizar programação lógica como forma de especificação de *pointcuts*. A sugestão parece surgir naturalmente da semelhança sintática e semântica entre o uso de *pointcuts* e de fatos em uma base Prolog.

3.2.1. Andrew

Em [7], Gybels propõe uma linguagem de definição de *pointcuts* baseada na linguagem de programação lógica QSOUL. Esta linguagem é uma variante simplificada de Prolog, desenhada para permitir meta-programação sobre programas em Smalltalk. A linguagem orientada a objetos Smalltalk, por sua vez, serve de base para a implementação da parte funcional da linguagem orientada a aspectos proposta pelos autores, denominada Andrew.

O modelo de pontos de junção de Andrew é composto de predicados que refletem as poucas construções primitivas de Smalltalk, como envios de mensagens e atribuições de valores a variáveis. A partir desses pontos de junção primitivos, os autores constroem *pointcuts* mais complexos aproveitando-se de convenções utilizadas por programadores dessa linguagem, já que ela não provê sintaxes específicas para algumas funcionalidades como tratamento de exceções e inicialização de instâncias.

Esta possibilidade de construir *pointcuts* para eventos que não correspondem a construções primitivas da linguagem é citada pelos autores como uma de suas motivações.

No trabalho [8], Gybels e Brichau elencam as características da sua linguagem (principalmente derivadas de seu paradigma lógico) que permitem a definição de *pointcuts* mais expressivos. Estas características são:

a) Unificação de predicados: permite que sejam feitas comparações (“matching”) entre as propriedades dos pontos de junção, além de produzir resultados que podem ser usados em outras comparações;

b) Processamento sobre propriedades: provê a capacidade de derivar valores calculados sobre propriedades, por exemplo, através de cálculos algébricos;

c) Ligação com “sombras”: as “sombras” são os pontos da estrutura estática de um programa ao qual correspondem pontos na execução do mesmo. A linguagem Andrew possui um modelo estático e um dinâmico, permitindo o uso de ambos na composição de *pointcuts*.

d) Regras parametrizadas reutilizáveis: as regras das linguagens lógicas possuem variáveis que podem ser usadas como restrições ou como resultados, permitindo sua reutilização de forma genérica;

e) Recursão: a recursão torna esta linguagem computacionalmente completa, permitindo capturar padrões recursivos.

As características a), b), d) e e) estão ligadas diretamente ao paradigma lógico escolhido, enquanto a característica c) faz parte do modelo de dados desenvolvido pelos autores.

3.2.2. Gamma

Um outro trabalho, mais recente, que utiliza uma linguagem lógica para definição de *pointcuts* está em [13]. Os autores propõem uma linguagem denominada Gamma, uma combinação da linguagem OO "de brinquedo" L2 (um pequeno subconjunto de Java) para a implementação da parte funcional com Prolog para a definição de *pointcuts*.

A principal característica da linguagem proposta é o modelo de dados sobre o qual a definição de *pointcuts* é desenvolvida. Este modelo baseia-se em um traçado ("trace") da execução do programa (portanto um modelo dinâmico) em que cada evento está associado a um número seqüencial, chamado de *timestamp*. Todos os predicados utilizados têm como primeiro argumento um *timestamp*. Isto permite que se façam relações temporais entre pontos de junção para compor os *pointcuts*, inclusive em relação a eventos no futuro de uma determinada linha de execução.

Esta flexibilidade permite definir *pointcuts* complexos, como o conhecido `cflow` do Aspectj, de forma sucinta:

```
% T2 está no fluxo de controle da chamada em T1
cflow(T1, T2) :-
  calls(T1,_,_,_,_),
  endcall(T3,T1,_),
  isbefore(T1,T2),
  isbefore(T2,T3).
```

A definição acima pode ser lida como: dados dois instantes T1 e T2, o evento ocorrido em T2 está no fluxo de controle do evento em T1 se T1 é anterior a T2, o evento em T1 é uma chamada de método, e a finalização da chamada iniciada em T1 ocorre em um momento T3, posterior a T2.

O exemplo abaixo mostra o uso de eventos no futuro para a definição de um *pointcut*:

```
before calls(Now,server,_,execute,_),
    cflow(Now,T),
    calls(T,database,_,protected,_) {
  this.db.authenticate(true)
}
```

A intenção é que, antes que sejam invocados métodos protegidos, seja forçada uma autenticação do usuário. Esta situação é semelhante à apresentada na seção 2.1.2.

Os autores chamam à atenção para a possibilidade de criar aspectos paradoxais. Como um *pointcut* pode depender de eventos posteriores à aplicação do conselho (*advice*) associado a ele, o conselho pode alterar as condições que o dispararam, fazendo com que ele não devesse ter sido disparado.

3.2.3. Alpha

Em [16], os mesmos autores buscam tornar mais prática a sua proposta anterior, restringindo condições e detalhando a implementação. Eles propõem a linguagem

Alpha, similar a Gamma exceto por não permitir referências a eventos no futuro, além de dispor de mais modelos de dados sobre os quais se pode compor *pointcuts.*

Os autores propõem o uso de quatro diferentes e complementares fontes de informação: uma representação da árvore abstrata de sintaxe, uma representação do armazenamento de objetos (*heap*), uma representação do tipo estático de cada expressão do programa, e uma representação do traçado (*trace*) da execução do programa.

É apresentada uma comparação interessante entre diversas formas de definir o mesmo *pointcut*, para o exemplo do editor de figuras. A comparação é feita em termos da capacidade das definições de resistir a mudanças no programa-base.

Abaixo, as diversas formas apresentadas para implementar o *pointcut* em questão em Alpha, em ordem de qualidade:

```
class DisplayUpdate extends Object {
  Display d;

  // enum pointcut
  after set(P,x,_); set(P,y,_); set(P,'start',_); set(P,'end',_),
    instanceof(P,'FigureElement') { this.d.draw(P); }

  // set* pointcut
  after set(P,_,_), instanceof(P,'FigureElement') { this.d.draw(P); }

  // pcflow pointcut
  after now(ID), set(ID,ExpID1,P,F,_), instanceof(P,'FigureElement '),
    pcflow(Display,'drawAll',(_,get((ExpID2,_),F))),
    hastype(ExpID2,'FigureElement') { this.d.draw(P); }

  // cflow pointcut
  after set(P,F,_), get(T1,_,P,F,_), mostRecent(T2,calls(T2,_,@this.d,'drawAll',_)),
    cflow(T1, T2), instanceof (P, 'FigureElement') { this.d.draw(P); }

  // cflowreach pointcut
  after set(P,F,_), get(T1,_,P,F,_), mostRecent(T2,calls(T2,_,@this.d,'drawAll',_)),
    cflow(T1,T2), reachable(Q,P), instanceof(Q,'FigureElement') { this.d.draw(P); }

}
```

As duas primeiras definições são do estilo de definição por enumeração (no primeiro caso de todos os pontos de junção, no segundo usando máscaras para selecionar um conjunto).
As três últimas buscam definições semânticas do *pointcut*, sempre partindo da idéia de capturar atribuições a campos que sejam lidos durante a execução do método `drawAll`, que faz a atualização da exibição das figuras.

No caso do `pcflow`, é utilizada uma estimativa do fluxo de controle da chamada ao método `drawAll` com base na estrutura estática do programa. Para o `cflow`, é utilizada a informação de tempo de execução dessa chamada, melhorando a precisão do *pointcut*. Por fim, o *pointcut* cflowreach busca as atribuições feitas dentro do fluxo de controle da chamada a `drawAll`, a campos de objetos aos quais o `FigureElement` em questão tenha acesso por meio de referências.

Os autores argumentam que esta última forma de definição é a mais resistente a mudanças, pois seria capaz de resistir às seguintes classes de alterações: refatoramento que coloca parte do estado de um derivado de FigureElement fora da

própria classe; adição de um novo derivado de FigureElement à hierarquia; mudança das condições em que um redesenho de tela é necessário; adição ou remoção de um campo cuja alteração torna um redesenho necessário; e adição ou remoção de um campo à classe que não afeta a operação de redesenho.

Na seção de trabalho futuro, os autores afirmam que pretendem adaptar sua tecnologia a uma linguagem de programação de verdade, e superar os obstáculos de desempenho para tornar a sua abordagem utilizável na prática.

3.3. Programação Imperativa

3.3.1. Josh

Em [5], é introduzida uma nova linguagem de programação chama Josh. Esta linguagem possui os mesmos recursos do AspectJ, exceto pela habilidade de definir novos *pointcuts* primitivos.

A definição desses *pointcuts* é feita na mesma linguagem que serve de base para a parte orientada a objetos e para a implementação dos conselhos, Java.

A base para a implementação e também para a filosofia de Josh é um arcabouço de meta-programação e manipulação de *bytecodes* Java chamado Javassist. O Javassist serve como API para a análise da estrutura do programa por parte das implementações de *pointcuts*, e também como ferramenta para introdução dos conselhos em meio ao programa orientado a objetos.

Graças a esta última característica, Josh oferece um recurso curioso, não encontrado em outras linguagens orientadas a aspectos: construir dinamicamente (no momento da "tecelagem" dos aspectos) trechos de código Java, utilizando valores obtidos por meta-programação. Esta característica, entretanto, não está relacionada à definição de *pointcuts* e, portanto, não é objeto deste trabalho.

Segue um exemplo de definição de um novo *pointcut* primitivo em Josh:

```
static boolean updater(MethodCall mc,
    String[] args, JoshContext jc) {

  CtClass root = jc.getCtClass(args[0]);
  String mname = args[1];
  CtMethod mth = mc.getMethod();

  // skip if the method is redraw().
  if (root.getName().equals(mname))
    return false;

  Hashtable fields = enumerateFields(jc, root, mname) ;
  updated = false;
  mth.instrument (new ExprEditor() {
    public void edit(FieldAccess expr) {
      String name = expr.getFieldName() ;
      if (expr.isWriter() && fields.get(name) == expr.getCtClass())
        updated = true;
    }
  });
  return updated;
}

Hashtable enumerateFields(JoshContext jc, CtClass root, String mname) {
  ...

  m.instrument (new ExprEditor() {
```

```
    public void edit(FieldAccess expr) {
      if (expr.isReader() && expr.getCtClass().subclassOf(root))
        fields.put(expr.getFieldName(), expr.getCtClass());
    }
  });

  ...
}
```

Este *pointcut* pode ser utilizado para resolver o problema da seção 2.1.1, que ficaria semelhante ao pseudo-código a seguir:

```
aspect DisplayUpdating {
 pointcut move() :
  call(updater(void FigureElement+.*(..)));

 after() returning :
   move() && target(fe) {
  fe.redraw();
 }
}
```

Embora o uso do novo *pointcut* seja simples, a sua definição é muito mais complicada do que seria nas abordagens que utilizam linguagens lógicas ou funcionais. Além disso, Josh leva em conta apenas informação estática, disponível no momento da "tecelagem" dos aspectos. Isto significa que uma atualização da exibição, no exemplo, poderia ser disparada sem que uma variável relevante tivesse sido alterada (por exemplo, caso haja um condicional no caminho).

Do ponto de vista prático, esta abordagem parece ser a mais próxima da realidade. A implementação do protótipo teve desempenho muito próximo ao de linguagens mais maduras, como o AspectJ, e causou pequeno sobrepeso em comparação com o programa sem a aplicação de aspectos.

3.4. Resumo comparativo

Nesta seção, apresentamos um resumo comparativo e crítico das propostas analisadas no presente trabalho.

A tabela abaixo é uma tentativa de classificação crítica das linguagens propostas. A seguir, detalharemos as informações e análises nela expressas.

Linguagem	**Paradigma**	***Pointcuts* dinâmicos?**	**Informação temporal?**	**Resistência a mudanças**	**Clareza de intenção**	**Viabilidade prática**
XQuery / BAT	Funcional	não	não	média	alta	média
Andrew	Lógico	sim	não	média	alta	média
Gamma	Lógico	sim	sim	alta	alta	baixa
Alpha	Lógico	sim	sim	alta	alta	média
Josh	Imperativo	não	não	média	média	alta

3.4.1. Paradigma

Neste item é informado o paradigma de programação da linguagem de definição de *pointcuts*, conforme declarado pelos respectivos autores.

3.4.2. *Pointcuts* dinâmicos

Por *pointcuts* dinâmicos, entende-se a capacidade de definir *pointcuts* com base em informações que estão disponíveis apenas em tempo de execução do programa.

As linguagens XQuery/BAT e Josh não provêm esta possibilidade, considerando-a como trabalho futuro.

3.4.3. Informação temporal

Indica-se aqui se a linguagem permite que *pointcuts* sejam compostos com base em informação dinâmica (portanto implicando um sim ao item 3.4.2) de eventos ocorridos no passado ou futuro da execução do programa.

Apenas as linguagens Gamma e Alpha provêem este recurso. No caso de Alpha, apenas para eventos no passado, e para Gamma também no futuro.

3.4.4. Resistência a mudanças

Neste item é feita uma avaliação da capacidade de resistência a mudanças dos *pointcuts* que podem ser escritos na linguagem. A base de comparação, considerada baixa (que todos pretendem superar), é a linguagem AspectJ e outras semelhantes que estão disponíveis atualmente (conforme exposto na seção 1.2).

As linguagens XQuery/BAT e Josh foram consideradas médias neste quesito. No caso de Josh e do XQuery/BAT, isto se deve à ausência de informação de tempo de execução, que limita a expressividade. No caso de Andrew, que possui modelo de dados dinâmico, a limitação é a ausência de informação sobre eventos passados e futuros.

Finalmente, as linguagens Gamma e Alpha são consideradas capazes de gerar *pointcuts* com alta resistência a mudanças, pelos motivos expostos nas seções que as detalham.

3.4.5. Clareza de intenção

A clareza de intenção procura avaliar o quanto o *pointcut* captura e transmite a real intenção do programador que o criou.

A única linguagem que foi considerada média neste aspecto é Josh. Apesar de permitir capturar de forma razoavelmente clara a intenção, o paradigma imperativo combinado ao arcabouço de manipulação de *bytecode* relativamente de baixo nível produzem definições que são longas e de entendimento mais difícil do que nos outros casos.

3.4.6. Viabilidade prática

Entre as linguagens apresentadas, a que parece ser mais viável na prática é Josh, por ser baseada em um arcabouço que já é base para outras implementações de programação orientada a aspectos [9]. Além disso, não exige que seja integrada uma linguagem diferente da que é usada para desenvolver o programa sobre o qual os aspectos devem ser aplicados.

A linguagem Gamma é considerada de baixa viabilidade, devido aos desafios apresentados por sua arquitetura que permite analisar eventos futuros. Isto foi apontado pelos próprios autores, que procuraram simplificar sua proposta na linguagem Alpha.

4. Conclusões

O campo da programação orientada a aspectos poderia ser grandemente beneficiado pela possibilidade de definir *pointcuts* de forma aberta e flexível. Como vimos, os *pointcuts* semânticos são a maneira mais flexível e precisa de aplicar aspectos, e eles dependem de recursos que não estão à disposição nas linguagens orientadas a aspectos dominantes atualmente.

Os trabalhos apresentados fazem progressos nesta direção, mas está claro que o problema está ainda longe de ser resolvido. Principalmente no que se refere a *pointcuts* dinâmicos (que dependem de condições conhecidas apenas durante a execução do programa, ou às vezes depois dela), existe grande dificuldade de implementar soluções eficientes e com modelos de programação de complexidade aceitável. Alguns dos problemas envolvidos são inclusive NP-completos [15].

Existem, portanto, três direções principais nas quais é preciso progredir para fazer dos *pointcuts* abertos uma realidade. A primeira delas é definir conjuntos de linguagem e meta-modelo que sejam ao mesmo tempo expressivos e de fácil utilização. A segunda é criar implementações eficientes desses modelos de programação. Por fim, é preciso evoluir no sentido dos *pointcuts* dinâmicos, buscando algoritmos eficientes para implementá-los.

5. Referências

[1] **Aspectj project**. http://eclipse.org/aspectj/.

[2] AspectJ Team, Xerox Corporation. **AspectJ Programming Guide**. http://eclipse.org/aspectj/doc/released/progguide/index.html. 2005.

[3] AspectWerkz. http://aspectwerkz.codehaus.org.

[4] BRICHAU, Johan; HAUPT, Michael (editores). **Survey of Aspect-oriented Languages and Execution Models**. European Network of Excellence in AOSD. Maio/2005.

[5] CHIBA, Shigeru; NAKAGAWA, Kiyoshi. **Josh: An Open AspectJ-like Language**. In AOSD'04, março/2004, Lancaster, Inglaterra.

[6] EICHBERG, Michael; MEZINI, Mira; OSTERMAN, Klaus. **Pointcuts as functional queries**. In The Second ASIAN Symposium on Programming Languages and Systems (APLAS 2004), Springer, LNCS.

[7] GYBELS, Kris. **Using a logic language to express cross-cutting through dynamic joinpoints**, Second Workshop on Aspect-Oriented Software Development of the GI, Bonn, February 21-22, 2002.

[8] GYBELS, Kris; BRICHAU, Johan. **Arranging Language Features for More Robust Pattern–Based Crosscuts**. In Proceedings of AOSD 2003, páginas 60–69, Boston, Massachusetts. ACM Press.

[9] JBoss AOP. http://www.jboss.org/products/aop.

[10] KICZALES, Gregor. **Keynote talk at AOSD 2003**. http://www.cs.ubc.ca/~gregor/papers/kiczales-aosd-2003.ppt.

[11] KICZALES, Gregor; LAMPING, John; MENDHEKAR, Anurag; MAEDA Chris; Lopes, CRISTINA V.; LOINGTIER, Jean-Marc; IRWIN, John. **Aspect-Oriented Programming**. In Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1997.

[12] KIRSTEN, Mik. **AOP tools comparison, Part 1: Language mechanisms**. IBM DeveloperWorks, Fevereiro/2005 (http://www-106.ibm.com/developerworks/java/library/j-aopwork1/).

[13] KLOSE, Karl; OSTERMANN, Klaus. **Back to the Future: Pointcuts as Predicates over Traces**. Workshop on Foundations of Aspect-Oriented Languages (FOAL'05), Chicago, USA, 2005.

[14] KOPPEN, Christian; STÖRZER, Maximilian. **PCDiff: Attacking the fragile pointcut problem** European Interactive Workshop on Aspects in Software (EIWAS), Setembro/2004.

[15] LIEBERHERR, Karl J.; PALM, Jeffrey; SUNDARAM, Ravi. **Expressiveness and Complexity of Crosscut Languages**. Workshop on Foundations of Aspect-Oriented Languages (FOAL'05), Chicago, USA, 2005.

[16] OSTERMANN, Klaus; MEZINI, Mira; BOCKISCH, Christoph. **Expressive Pointcuts for Increased Modularity**. European Conference on Object-Oriented Programming (ECOOP), Springer LNCS, 2005.