

# Understanding Aspects through Call Graph Enumeration and Pointcut Satisfiability

Jeffrey Palm, Pengcheng Wu, and Karl Lieberherr

Northeastern University, Boston MA 02115, USA,  
{jpalm,wupc,lieber}@ccs.neu.edu,

WWW home page: <http://www.ccs.neu.edu/research/demeter>

**Abstract.** An aspect-oriented program can be split into an aspect program and a target program to which these aspects are applied. The aspect program contains pointcuts that select *interesting* points in the target program. In an effort to componentize and better understand these aspects, we would like to reason about them in the absence of target programs. As a first step we present an algorithm that builds a representation of the set of call graphs to which a pointcut can apply – a problem called call graph enumeration. We also decide whether this set is non-empty – a problem called pointcut satisfiability. Solutions to these problems may help to componentize and assign useful types to aspect-oriented programs.

## 1 Introduction

Aspect-oriented program (AOP) provides a way to modularly separate concerns in programs by applying aspects to target programs [1, 2]. This paper presents two problems that we see as vital in the understanding and componentizing of AOP programs. In its purest form, an aspect quantifies over a target program, selecting *interesting* points in its execution, and specifies some action to take at those points. There have been many efforts to reason about AO programs in the context of target programs [3–7], but none to the knowledge of the authors that attempt to reason about aspects in the absence of target programs. This paper is a first step towards the latter goal.

We define two problems, which have analogies in related domains, such as object-oriented programming [8], and even unrelated domains, such as linguistics [9]. The first problem is, *given a pointcut, can we decide whether there is a call graph that could satisfy this pointcut?* The second problem is, *if this is true, could we enumerate these call graphs?* In doing so we present algorithm to solve these problems as well as give possible applications to these answers. Additionally, we show that a solution to the full problem is NP-complete.

### 1.1 Motivation

We motivate this work by noting the discussion about *library* or *plug-and-play* aspects in the AOP community. In general, these are notions of pre-packaged aspects that a programmer can plug into her program... and work! We believe that

componetizing aspects is vital, but until we have an understanding of aspects in the absense of target programs, we can never package them up as components. So, we claim that a step forward in reasoning about aspects is to reason about the programs on which they operate. We now give three ancillary applications:

- Solutions to these problems could help the question *what is the type of an aspect?* We believe a valid type for an aspect is the set of all call graphs on which it is applicable. This is analogous to Palsberg’s notion that a class graph inferred from an adaptive program is a valid typing for that program [8].
- Call graph enumeration could help aspect writers to visualize and reason about the aspects they are writing. That is, given an aspect, what are the applicable programs which would be valid for this aspect. Often a programmer writes an aspect that applies advice at unintended points or omits advice at intended points. This is related to componentizing, but is best summarized as reasoning about the applicability of aspects.
- Lastly, how can we test aspects without a notion of their applicability? Moreso, a tester would want to test the *edges-case* call graphs that just *barely* match a given pointcut. But without an understanding of an aspect’s applicability, what are these cases?

We hope to answer these questions and provoke additional interesting issues with this work.

## 2 The Model

In this section we give a brief explanation of the call graphs and our PCD language. In addition we introduce concepts used in the rest of the paper.

### 2.1 Call graphs

For our purposes programs are expressed as call graphs and a **call graph** is a directed graph  $G = (V, E)$  with nodes  $V$  and directed edges  $E$ . In addition, all nodes have labels from some alphabet  $\Sigma$  and these labels represent function names. We define  $\mathcal{N} : V \rightarrow \Sigma$  as a mapping from nodes in  $V$  to labels in  $\Sigma$ . The label for a node  $v \in V$  is denoted  $\mathcal{N}(v)$ . All edges are denoted  $(v, u)$  for  $v, u \in V$ , and paths are denoted  $(v_0, \dots, v_{n-1})$  for  $n$  nodes in  $V$  and is the shorthand for  $(v_0, v_1), (v_1, v_2) \dots (v_{n-2}, v_{n-1})$ . These edges represent function calls. Our language is model is simpler than that of languages, such as AspectJ [10], that operate on languages with polymorphism and dynamic, such as Java [11]. We can statically determine all call sites and made this decision knowing that our ideas could be expanded to more complex languages.

Nodes without incoming edges are contained in the set  $\text{Source}(G)$ , and nodes without outgoing edges are contained in the set  $\text{Sink}(G)$ . These can also be applied to paths.  $\text{Reach}_G(v)$  is the set of all reachable nodes from  $v$  including  $v$ , and  $\text{Pred}(v)$  is the set of predecessors of  $v$  including  $v$ . Without loss of generality

we assume there is only one *main* function in the program which is the source. Lastly, we define a **normal directed graph** to be a directed graph with one source, one sink, and one out-going edge for each interior node.

## 2.2 Join points and advice

Advice, in its most general sense, is some action to take at a certain point in a program. Join points are the events during the execution of the program at which advice may run. Wand *et al.* give three *kinds* of join points – **pcall**, **pexecution**, and **aexecution** – coresponding to procedure calls, procedure execution, and advice execution [3]. We use one type – function call.

## 2.3 Pointcut designators

A PCD is a formula that specifies the set of join points to which a piece of advice is applicable. The language of PCDs is given below where  $n \in \Sigma$  and  $p$  are PCDs. These minimal PCDs consists of *calls*, *cflow*, and boolean operators found in AspectJ [10] and other AOP languages.

### PCD syntax

$$p ::= \text{call}(n) \mid \text{cflow}(p) \mid p_1 \vee p_2 \mid p_1 \wedge p_2 \mid \neg p$$

**PCD matching** Wand *et al.* give a semantics of PCDs by a function *match-pcd* that takes a PCD and join point[3]. We use an adaptation whose meaning is the function  $M : P \times G \rightarrow \text{Paths}_G$ , which takes a PCD  $p$  and call graph  $G$  and returns the set of matching paths in  $G$ . The set  $\text{Paths}_G(A, B)$  consists of all paths from  $A$  to  $B$  in the graph  $G$  [12]. Additionally, if  $P_1$  and  $P_2$  are paths in  $G$  where all paths in  $P_1$  have target  $v$  and all paths in  $P_2$  have source  $v$ , we define

$$P_1 \circ P_2 = \{p \mid p = p_1 p_2 \text{ where } p_1 \in P_1 \text{ and } p_2 \in P_2\}$$

So, we the meaning of a PCD  $p$  wrt to a call graph  $G$  with source *main*, denoted  $M(p, G)$ , is defined by the following reductions.

### PCD semantics

$\text{CALL}$ $\text{call}(n) \Downarrow \text{Paths}_G(\text{main}, n)$			
$\text{CFLOW}$ $\frac{p \Downarrow S}{\text{cflow}(p) \Downarrow \bigcup_{s,t} \text{Paths}_G(\text{main}, s) \circ \text{Paths}_G(s, t), \text{ where } s \in S \text{ and } t \in \text{Reach}_G(s)}$			
<table style="width: 100%; border-collapse: collapse;"> <tr> <td style="text-align: center; padding: 5px;"> <math display="block">\text{AND}</math> <math display="block">\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \wedge p_2 \Downarrow S_1 \cap S_2}</math> </td> <td style="text-align: center; padding: 5px;"> <math display="block">\text{OR}</math> <math display="block">\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \vee p_2 \Downarrow S_1 \cup S_2}</math> </td> <td style="text-align: center; padding: 5px;"> <math display="block">\text{NOT}</math> <math display="block">\frac{p \Downarrow S}{\neg p \Downarrow \overline{S}}</math> </td> </tr> </table>	$\text{AND}$ $\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \wedge p_2 \Downarrow S_1 \cap S_2}$	$\text{OR}$ $\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \vee p_2 \Downarrow S_1 \cup S_2}$	$\text{NOT}$ $\frac{p \Downarrow S}{\neg p \Downarrow \overline{S}}$
$\text{AND}$ $\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \wedge p_2 \Downarrow S_1 \cap S_2}$	$\text{OR}$ $\frac{p_1 \Downarrow S_1, p_2 \Downarrow S_2}{p_1 \vee p_2 \Downarrow S_1 \cup S_2}$	$\text{NOT}$ $\frac{p \Downarrow S}{\neg p \Downarrow \overline{S}}$	

Given a call graph  $G$ ,  $M(\text{call}(n), G)$  is all paths from  $\text{main}$  to  $n$ ;  $M(\text{cflow}(p), G)$  is a path set consisting of all paths obtained from appending those in  $M(p, G)$  to every path reachable from the target of  $M(p, G)$ ; the three boolean operators translate to corresponding set operations introduced in the Demeter System for path sets [13]; and the complement of a path set  $\overline{P}$  is taken

#### 2.4 The problems

Having defined PCD matching, we can now formally state our definition of PCD satisfiability and call graph enumeration.

**Definition 1 (PCD satisfiability).** *A PCD  $p$  is satisfiable if there exists a call graph  $G$  such that  $M(p, G)$  is non-empty.*

We call this problem PCDSAT. In the same manner, we can define call graph enumeration in terms of PCD satisfiability.

**Definition 2 (Call graph enumeration).** *Given a PCD  $p$  a call graph enumeration is all  $G$  that satisfy  $p$ .*

#### 2.5 Aspects

Lastly, in our model aspects are represented as sets of pointcuts. Without loss of generality, any results gained from one pointcut can be applied to a an aspect containing more than one.

### 3 Algorithm

We've stated two purposes and this algorithm addresses them both. The intuition of our algorithm is to reduce the problem of finding satisfiable call graphs of

pointcuts to the problem of finding satisfiable directed graphs on which we place constraints. These constraints restrict the labels we place on nodes and the connectivity of nodes. If we can construct a call graph that contains at least one path that does not violate any of these constraints, then we have shown the existence of a path satisfying our PCD. A graph with at least one path satisfying a PCD can then be elaborated with additional edges and nodes to represent all possible call graphs satisfying that PCD. Thus, we will find a solution to the satisfiability problem on our way to enumerating possible call graphs.

Our general approach is to, first, convert each pointcut into a negation normal form and then into a disjunctive normal form as presented in [14]. At this point we analyze each clause of the disjunction and try to construct a minimal call graph from which any satisfying solution can be generated; these will be called *solved forms*. Then, the outcome of this algorithm is the set of solved forms, and each solved form represents a minimal call graph satisfying the pointcut.

### 3.1 NP-completeness proof

Having formally defined the problem of PCD satisfiability in Section 2.4 and call graph enumeration in terms of the latter, we will now show both these problems NP-complete. In the usual way we give a polynomial-time verifier and then show a reduction from 3SAT. For this proof, we assume, without loss of generality, PCDs are conjunctive normal forms where each clause is of the form  $(x_1 \wedge x_2 \wedge x_3)$  and  $x_i$  is either  $\text{cflow}(\text{call}(n))$  or  $\neg\text{cflow}(\text{call}(n))$ .

*Proof.* First, we show PCDSAT is in NP. Given a call graph  $G$  and PCD  $p$  in the form of (??) we construct a verifier with a path of nodes,  $c = (v_0, v_1, \dots, v_n)$ , as the certificate. So our verifier, on input  $\langle\langle G, p \rangle, c\rangle$ , does the following:

1. Test whether all  $v \in c$  are nodes in  $G$ .
2. Start with an empty boolean formula  $\phi = \text{true}$ .
3. For every clause of  $p$ ,  $(x_1 \vee x_2 \vee x_3)$ , add a corresponding clause,  $(y_1 \vee y_2 \vee y_3)$  to  $\phi$  where  $y_i = x_i$  if  $x_i$  is of the form  $\text{cflow}(\text{call}(x_i))$ ; otherwise  $y_i = \neg x_i$ .
4. Initially set all terms in  $\phi$  to *false*.
5. For every  $v$  in  $c$ , set  $v = \text{true}$  in  $\phi$ .
6. Solve  $\phi$ .

This is clearly polynomial, so PCDSAT is in NP.

We now show 3SAT is polynomial-time reducible to PCDSAT using the reverse of the construction used in the verifier. That is, for a boolean formula

$$\phi = (a_1 \vee b_1 \vee c_1) \wedge (a_2 \vee b_2 \vee c_2) \wedge \dots \wedge (a_n \vee b_n \vee c_n),$$

We create a PCD  $p$  in conjunctive normal form so that for every clause in  $\phi$ ,  $(y_1 \vee y_2 \vee y_3)$  we add a clause,  $(x_1 \vee x_2 \vee x_3)$  to  $p$  where  $x_i = \text{cflow}(\text{call}(y_i))$  if  $y_i$  is of the form  $a$ ; otherwise  $x_i = \neg\text{cflow}(\text{call}(y_i))$ . Clearly this reduction is polynomial time. So, if PCDSAT were in P and had a deterministic polynomial time algorithm we could solve 3SAT in polynomial time. Thus, PCDSAT is NP-complete.

### 3.2 Reduction of PCDs to negation normal forms

As stated previously, we consider PCDs in a restricted form. So our first step is to use Wu and Lieberherr's reduction rules to reduce a PCD into a form where the  $\neg$  operator can only be used in one of the following forms:  $\neg\text{call}(n)$ ,  $\neg\text{cflow}(\text{call}(n))$  and  $\neg\text{cflow}(\neg\text{call}(n))$ . This is called *negation normal form*. Additionally, we assume no PCD contains a terms of the form  $a_1 \wedge a_2 \wedge \dots \wedge a_n$  with  $n \geq 2$  where for  $i, j \leq n$  and  $i \neq j$ ,  $a_i = \text{call}(n)$  and  $a_j = \text{call}(n)$ , because these terms can be trivially reduced to  $\text{call}(n)$  [14]. Wu also provides a proof that any PCD can be reduced in this fashion [15]. Having transformed the PCD into negation normal form, we further reduce to a *disjunctive normal form*. A PCD  $p$  is in disjunctive normal form if:

1.  $p$  is in negation normal form, and
2.  $p$  is in the form  $p_1 \vee p_2 \vee \dots \vee p_n$  where each  $p_i$  is in the following *conjunctive normal form*:

$$q \wedge \text{cflow}(r) \wedge \neg\text{cflow}(s_1) \wedge \dots \wedge \neg\text{cflow}(s_k)$$

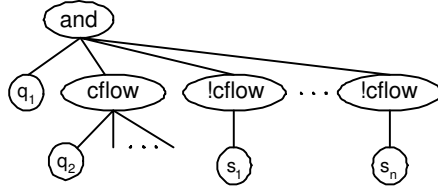
where

- (a)  $q$  is  $\text{call}(n)$  or  $\neg\text{call}(n)$  or an *empty* term, and
- (b)  $r$  is in conjunctive normal form
- (c)  $s_i$  are  $\text{call}(n)$  or  $\neg\text{call}(n)$ . This is upheld by our reduction to negation normal form.

Each  $p_i$  is a conjunctive form (CF) term [14].

For the rest of the paper, every PCD is assumed to be in its disjunctive normal form. It is also evident from the form of CFs that PCDs can be represented in tree form where call terms and empty terms are terminal symbols. So we consider abstract syntax of PCDs in the form of Figure 1 for the expression:

$$\text{call}(q_1) \wedge \text{cflow}(\text{call}(q_2) \wedge \dots) \wedge \neg\text{cflow}(\text{call}(q_3) \wedge \dots) \dots \wedge \neg\text{cflow}(\text{call}(q_n) \wedge \dots)$$



**Fig. 1.** Example abstract syntax for a PCD tree.

### 3.3 Graph constraints

In addition to constructing a graph from PCDs, we will add constraints to that graph to restrict the paths of this graph that can satisfy for our PCD. These come in two flavors: **labelling constraints** restrict the label which we assign a node and **dominance constraints** restrict the labels which we can assign to node's predecessors.

**Labelling constraints** To every node  $v$  we attach a set of node labels that represent the labels which cannot be assigned to  $v$ . This set, called the **labelling constraint set** is denoted  $\mathcal{L}(v)$  and must satisfy  $\mathcal{N}(v) \notin \mathcal{L}(v)$ . That is, we cannot assign  $v$  any label found in  $\mathcal{L}(v)$ . These constraint are used to uphold the meaning of  $\neg\text{call}(n)$  terms.

**Dominance constraints** To every node  $v$  we, also, attach a set of node labels that together express the constraint that no node in  $\text{Pred}(v)$  may have a label in this set. This set, called the **dominance constraint set** is denoted  $\mathcal{D}(v)$ . In general, it must be true that there exists a path  $(v_0, v_1, \dots, v_n, \text{Sink}(G))$  so that for every  $u$  in this path  $\mathcal{N}(v_0) \notin \mathcal{D}(u)$ . But, since all constructed graphs are normal (as we show in Section 3.4), this is equivalent to  $\forall p \in \text{Reach}_G(v). \mathcal{N}(v) \notin \mathcal{D}(p)$ . That is, the label of  $v$  is not contained in the dominance constraint of any reachable node.<sup>1</sup>

Before combining these two constraints, note that, as we pointed out in Section 2.1, all nodes of valid call graphs are labelled. We need to ensure that there actually does exist a valid labelling for a given graph. So we say the valid labels for a labelled node is a singleton set of that node's label, and a valid label for unlabelled node  $v$ , denoted  $\text{ValidLabels}(v)$ , is all those not found in reachable nodes' dominance constraints.

#### Valid labels

$$\text{ValidLabels}(v) = \begin{cases} \Sigma - \bigcup_{u \in \text{Reach}_G(v)} \mathcal{D}(u) & \text{for unlabelled } v \\ \{\mathcal{N}(v)\} & \text{for labelled } v \end{cases}$$

We combine these labelling and dominance constraint sets together to form the following predicate which must be true for all nodes:

<sup>1</sup> The term *dominance constraints* is used in the computational linguistics field to refer to logical tree descriptions. We reuse this term for our constraints because both have the same flavor. But it should be noted that the dominance constraints found in computational linguistics (for example [16]) in general are different.

### Constraint predicate

$$\begin{aligned}
\mathcal{P}(v) = & \quad \forall p \in \text{Reach}_G(v). \mathcal{N}(v) \notin \mathcal{D}(p) && \text{(Dominance)} \\
& \wedge \mathcal{N}(v) \notin \mathcal{L}(v) && \text{(Labelling)} \\
& \wedge \text{ValidLabels}(v) \neq \emptyset && \text{(Valid Labels)}
\end{aligned}$$

Now that we have a notion of constraints, we introduce the graph interpretation of a PCD, which consists of a normal directed graph and a constraint predicate. We call this a *solvable form* and denote it  $G \times \mathcal{P}$ . We use *solvable* and not *solved*, because there is still a chance that constraints in  $\mathcal{P}$  are violated by nodes in  $G$ .

### 3.4 Solvable form construction

We will now show how to construct a solvable form  $G \times \mathcal{P}$  from a PCD  $p$  in tree form. Intuitively, we are turning the PCD tree upside down and *carving* out the *call* nodes (or empty nodes) that don't appear under  $\neg\text{cflow}$  nodes. We then add a then add a source node, labelled *main* and an edge from *main* to the root of this carved out path. The dominance constraint set for a call or empty node consists of *call* nodes beneath all sibling  $\neg\text{cflow}$  nodes in the original tree. The labelling constraint set for *call* nodes will be empty and for  $\neg\text{call}(n)$  nodes will contain the singleton set  $\{n\}$ . To accomplish this we will search the PCD top-down and construct the graph bottom-up. We assume without loss of generality that each row of nodes in  $p$  are in the following order:

$$\neg\text{cflow}_1, \dots, \neg\text{cflow}_k, q, \text{cflow}$$

and perform a breadth-first search of  $p$  without searching into the  $\neg\text{cflow}$  nodes. We keep track of the previous node inserted in to  $G$  by the variable  $t$  initialized to *nil*. We start with an empty constraint predicate  $\mathcal{P}$ . For a given row do the following:

1. Initialize a node label set  $S \leftarrow \emptyset$ . For all  $\neg\text{cflow}(\text{call}(n))$  nodes,  $v$ , add  $n$  to  $S$ . Add  $\Sigma - n$  to  $S$  for  $\neg\text{cflow}(\neg\text{call}(n))$  nodes.
2. For  $q$  of the form  $\text{call}(n)$ , create a new node  $v$  with label  $n$ . For  $q$  of the form  $\neg\text{call}(n)$ , create a new node  $v$  with no label and labelling constraint  $\{n\}$ . If  $q$  is an empty node, create a new node  $v$  with node label and an empty labelling constraint. Add  $v$  to  $V$  and edge  $(v, t)$  to  $E$  if  $t \neq \text{nil}$ .
3. Add the dominance constraint set mapping  $v \rightarrow S$  to  $\mathcal{D}$ .
4. Search into the *cflow* node if one exists.

Finally, create a new node  $v_m$  with label *main*, add  $v_m$  to  $V$  and edge  $(v_m, t)$  to  $E$  if  $t \neq \text{nil}$ .

It is worth two things: First, for any edge  $(v, u)$ ,  $\mathcal{P}(u)$  is stronger than  $\mathcal{P}(v)$  because  $v \in \text{Pred}(u)$ . Also, our pointcut grammar clearly shows that there can

never be an empty node as the sink. So we can merge any empty node with its immediately reaching node, provided we add the empty node's dominance constraint to the reaching node. Secondly, from our construction this graph is clearly normal.

### 3.5 Satisfiability and solved forms

We have now constructed a solvable form  $G \times \mathcal{P}$  from PCD  $p$  and can define satisfiability of this pointcut in terms of the following problem:

**Definition 3 (Solvable form satisfiability).** *Given a solvable form  $G \times \mathcal{P}$ , label alphabet  $\Sigma$ , we say that  $G$  is satisfiable if  $\mathcal{P}(v)$  holds for all  $v \in V$ :*

That is, this is the problem of determining whether there are nodes in  $G$  that violate any constraints of any other nodes. Since we can obtain a normal directed graph from any pointcut, we can thus, define PCD satisfiability in terms of solvable form satisfiability. Now we say that if we can construct a solvable form  $G \times \mathcal{P}$  from PCD  $p$  and label alphabet  $\Sigma$ , PCD satisfiability of  $p$  is equivalent to the graph satisfiability of  $G$ .

Additionally, having shown this solvable form satisfiable, we can now call this tuple a **solved form**. In general we define a solved form as a solvable form  $G \times \mathcal{P}$  that is satisfiable.

### 3.6 Call graph construction

We now show how to construct a satisfying call graph from a solved form. whose graph  $G$  has possibly unlabelled nodes. So, for all nodes  $v$  without labels, we assign  $v$  a label such that  $\mathcal{P}(v)$  is not violated. Also, some nodes may have duplicate labels, so we need to combine all nodes with a particular label into one node. The outcome is a call graph  $G_{cg} = (V_{cg}, E_{cg})$ , and we proceed as follows:

1. Initialize  $V_{cg}$  and  $E_{cg}$  to empty. Add a node with label *main* to  $V_{cg}$ .
2. For all  $(u, v) \in E$  where  $l_u = \mathcal{N}(u)$  and  $l_v = \mathcal{N}(v)$ :
  - (a) If there is a node in  $V_{cg}$  with label  $l_u$ , let  $u'$  be that node. Otherwise create a new node  $u'$  with label  $l_u$  and add  $u'$  to  $V_{cg}$ .
  - (b) If there is a node in  $V_{cg}$  with label  $l_v$ , let  $v'$  be that node. Otherwise create a new node  $v'$  with label  $l_v$  and add  $v'$  to  $V_{cg}$ .
  - (c) If these does not exist an edge  $(u', v')$  in  $E_{cg}$  at this edge.

### 3.7 Call graph enumeration

Given a call graph  $G_{cg}$  constructed from a solved form, we can enumerate all possible call graphs derived from  $G_{cg}$  by repeating the following:

1. Add a new node.
2. Add an edge between existing nodes.
3. Add an edge from a new node.

4. Add an edge to a to new node.

Clearly, this process will enumerate all possible call graphs satisfying a CF, because we are just adding *additional* information, and this *additional* does not invalidate the existence of one path satisfying our PCD.

### 3.8 Correctness

We now give an argument why our solvable form construction is correct. Consider a CF clause in a disjunctive form:

$$q_1 \wedge \text{cflow}(q_2 \wedge \text{cflow}(\dots) \wedge \dots) \wedge \neg \text{cflow}(s_1) \wedge \dots \wedge \neg \text{cflow}(s_k)$$

For a call graph to satisfy this PCD  $q_1$  must be reachable from  $q_2$ . Our construction upholds this, because it traverses breadth-first (left to right) and insert  $q_1$  below  $q_2$ . Additionally, it must be the case that this path does not contain a node labelled  $n$  for every  $\neg \text{cflow}(\text{call}(n))$  term; this is upheld by the labelling constraints. It, also, must be the case that this path only contains a node labelled  $n$  for every  $\neg \text{cflow}(\neg \text{cflow}(n))$ . This, again, is upheld by our construction of the  $\{\Sigma - n\}$  labelling constraint.

### 3.9 Summary

Now that we have shown how to construct a solved form from a CF, we can conclude with the description of the algorithm in full. Given a PCD  $p$  convert this PCD into a set of CFs  $C = \{c_1, c_2, \dots, c_n\}$ . Start with an empty set  $S$ , and for every  $c \in C$ , construct a solved form  $s$ . If one such exists, add  $s$  to  $S$ . If  $S$  is empty we conclude that  $p$  is unsatisfiable. Otherwise,  $S$  contains all solved forms from which we enumerate any call graphs satisfying  $p$ .

## 4 Example

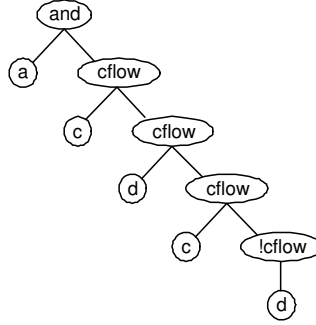
We now demonstrate our ideas with an example. Consider the PCD

$$\begin{aligned} p = & \text{call}(a) \\ & \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d))) . \\ & \wedge \text{cflow}(\text{call}(c) \wedge \text{cflow}(\text{call}(d))) \end{aligned} \quad (1)$$

Before proceeding we introduce some reductions that eliminate trivial contradictions and can make reductions much clearer:

#### PCD reductions

$$\begin{aligned} \text{call}(n) \wedge \neg \text{call}(n) & \rightarrow \textit{false} \\ \text{call}(n) \wedge \text{cflow}(\text{call}(n)) & \rightarrow \textit{false} \\ \text{cflow}(\text{call}(n)) \wedge \text{cflow}(\text{call}(n)) & \rightarrow \textit{false} \end{aligned}$$



**Fig. 2.** PCD tree obtained from the example PCD.

Because  $p$  is already in negation normal form, we begin by converting it to disjunctive normal form. In our first step we use the rule [14]:

$$cflow(p_1) \wedge cflow(p_2) \rightarrow cflow(p_1 \wedge cflow(p_2)) \wedge cflow(p_2 \wedge cflow(p_1))$$

to obtain

$$\begin{aligned} & \text{call}(a) \\ & \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(c) \wedge \text{cflow}(\text{call}(d)))) \\ & \wedge \text{cflow}(\text{call}(c) \wedge \text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d)))) \end{aligned}$$

Our reduction proceeds by reducing the clause inside the bottom  $cflow$ :

$$\text{call}(c) \wedge \text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d)))$$

to

$$\begin{aligned} & \text{call}(c) \wedge \\ & (\text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d)))) \vee \\ & \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(d))) \end{aligned}$$

Using our added reductions we can eliminate the bottom clauses, resulting in

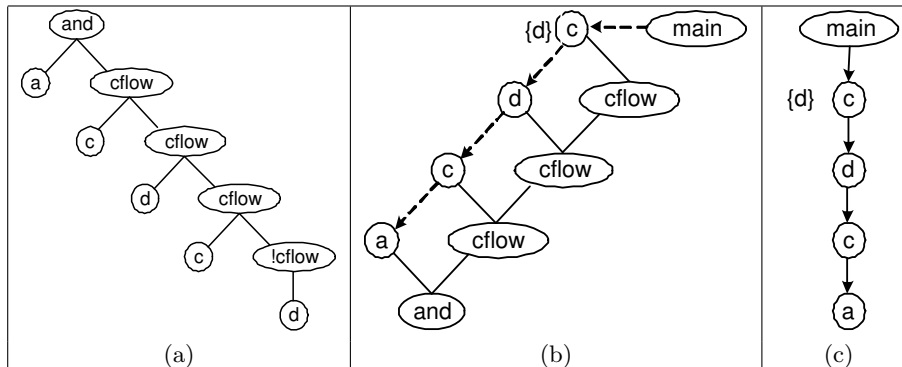
$$\text{call}(c) \wedge (\text{false} \vee \text{false})$$

which cannot be satisfied. So we are left with the CF in negation normal form:

$$\text{call}(a) \wedge \text{cflow}(\text{call}(c) \wedge \text{cflow}(\text{call}(d)) \wedge \text{cflow}(\text{call}(c) \wedge \neg \text{cflow}(\text{call}(d))))$$

Figure 2. show this CF in its AST form.

We, first, show our intuitive construction, as noted in Section 3.4, which is to invert the tree and *carve* out the terminal nodes. Figure 3 shows three stages of this construction. In these graphs, names inside nodes denote labels, and we do not show the node variables. (a) shows the original tree; (b) shows the inverted version where we have collected the negative constraints for the  $c$  node; and (c) shows the final graph.



**Fig. 3.** Intuitive solved form construction from the example PCD tree.

Figure 4 shows a walk-through of the construction using the BFS as described in Section 3.4. Gray nodes have been visited and the graph is constructed bottom-up on the right side of each frame. In (a)–(d) we simply visit nodes labelled  $a$ ,  $c$ ,  $d$ , and  $c$ ; we collect the negative constraints under the  $\neg cflow(d)$  node; and in (f) we attach a *main* node.

As Figure 4 (f) shows, we have obtained a normal directed graph and our dominance constraint map for  $\mathcal{P}$  contains one entry, which is  $v_2 \rightarrow \{d\}$ . In both constructions, we see that none of the node predicates are violated, so we proceed to construct a valid call graph. Walking this solved form to create the call graph is shown in Figure 6. Gray nodes, again, denote visited nodes, and names in the call graph nodes are labels, not variables.

Figure 6 shows example enumerations of our call graph in Figure 4 (f). We add a new node in (a), add an edge from a new node in (b), add an edge between existing nodes in (c), and add an edge to a new node in (d).

So, the solution to the PCD in (1) is the singleton set of our one solved form.

## 5 Related Work

We divide previous work into three categories: modularizing aspects, static analysis of aspects, and constructing graphs from specification.

### 5.1 Aspects and Modules

Dantas and Walker developed an AOP language called AspectML that allows programmers to control information hiding and access to the internals of a module through a simple type system [17]. This system is an extension to ML and upholds ML’s original module system. One key contribution of this work is that, unlike many AOP languages, AspectML allows separate compilation.

Lieberherr *et al.* point out that the flexibility introduced by AOP often compromises other favorable properties of programs, including modularity. In this

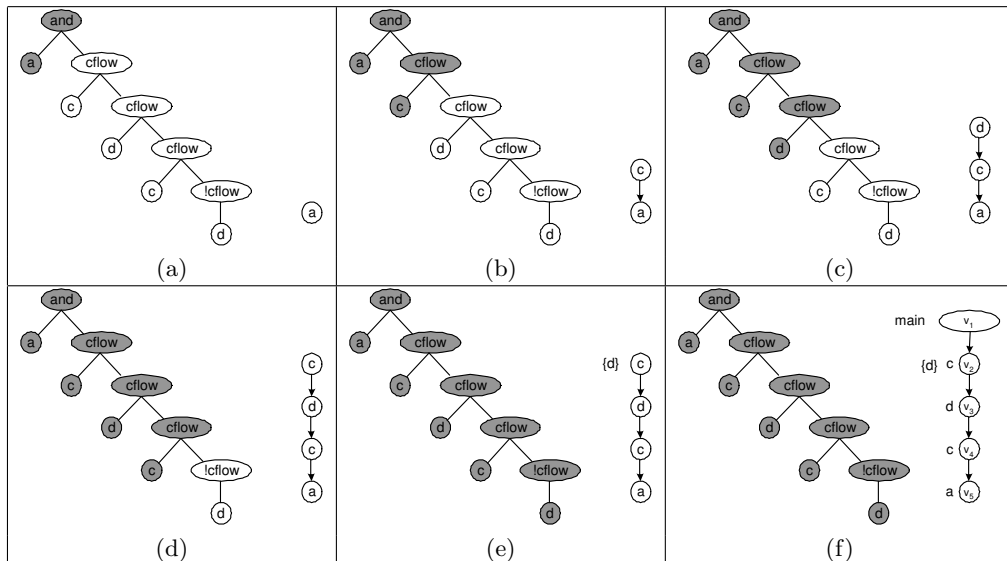


Fig. 4. Solved form from the example PCD tree.

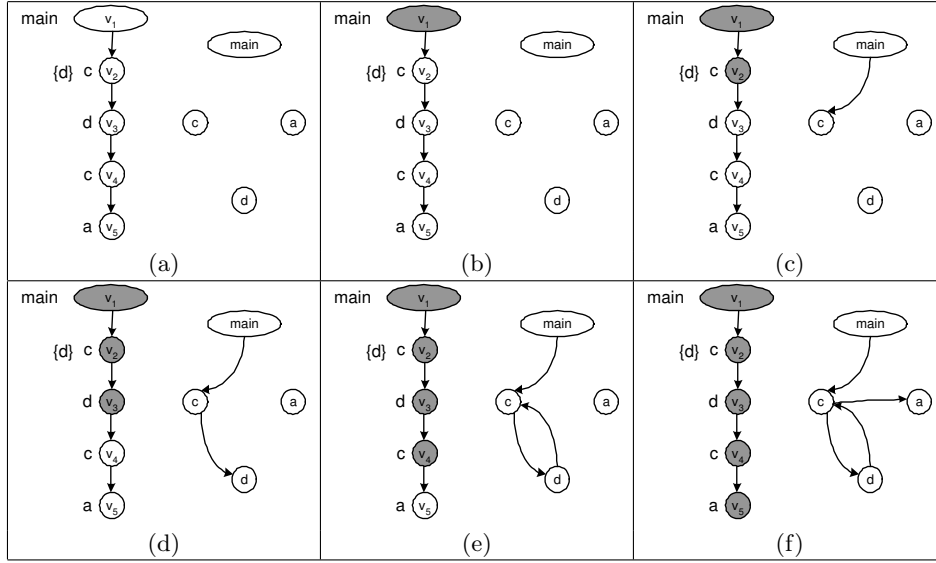
vain they introduce aspectual collaborations, which allow the expressing of aspects as part of modules in order to restore lost modular properties [18].

Jiazzi by McDirmid *et al.* allows the construction of large-scale binary components, called *units*, for Java [19] that can be thought of generalizations to Java packages with added support for external linking and compilation. Along these lines, the same project introduced AOP with Jiazzi where units could be thought of as aspects [20] with the ability to cleanly separate crosscutting concerns. This system preserved the Java-version’s external linking and separate compilation. Two components were central in accomplishing this: (1) Classes could be enhanced with methods and fields. (2) The signatures of methods and classes could be refined to allow more modularization.

## 5.2 Static analysis of aspects

Sereni and de Moor use a a more primitive syntax for pointcuts then found in common languages and show that this allows more efficient runtime matching [21]. We do not do static analysis in this same manner, but as our source and PCD languages become more complex, such analyses could prove useful. They also give an argument that the language of pointcuts is regular, which could be useful in constructing a more efficient algorithm. For, if the language of PCDs is in fact regular we could model our solutions as DFAs and construct them using the usual set operations.

Masuhara *et al.* present a semantics for compiling AOP programs using an operational semantic model [22]. They use partial evaluation to find locations



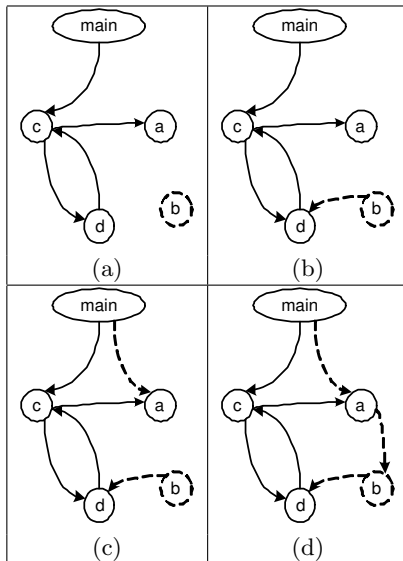
**Fig. 5.** Call graph construction for the example PCD tree.

to insert aspect code (advice), and in general, they investigate how to reason about the interactions between aspects and their target programs. As in Sereni’s work, these analyses could prove useful as we expand our work to more complex systems.

### 5.3 Graph construction

Palsberg gives a polynomial-time algorithm for constructing a class graph from an adaptive program [8]. This work posed questions such as “The fundamental question how flexible was a given software component?”, “with which architectures will the component work?”, and “do two components have compatible set of architectures with which they can work?”. In order to answer these questions, Palsberg infers class graphs (consisting of methods and fields) from adaptive programs. These adaptive programs contained *traversals* and *wrappers*, which are very similar to PCDs and advice. Indeed this work was influential on our’s, and introduced the concept that the type of an adaptive program was parametrized by the class graphs on which that program operated.

Althaus *et al.* give a polynomial-time algorithm for checking for satisfiability and enumerating solutions to dominance constraints [23]. In this work and computational linguistics in general, dominance constraints are logical descriptions of trees and are used to solve the underspecification problem [24]. In its simplest form this language is a conjunction of constraint placed on which nodes can dominate other nodes. Therefor the analogous problems Althaus and his community face are how to construct a tree that satisfies these constraints. There are



**Fig. 6.** Call graph enumeration for the example PCD.

two importances of this work: First, the satisfiability problem for trees and for pointcuts is very similar, and we took a similar approach as did Althaus' group. Lastly, they define a restricted form of dominance constraints, called *normal* dominance constraints, and a similar restriction to the pointcut language could prove useful in reduce our algorithm to P-space.

## 6 Future work

We are currently looking into the following extensions of this work:

- A polynomial-time algorithm for PCD satisfiability and call graph enumeration. Although the problem presented was interesting, our algorithm was exponential, and this is impractical in practice. In order to make our ideas mainstream we are investigating a poly-time algorithm, or if we can prove one such algorithm does not exist, we will move to an approximation.
- As we stated in Section 2.1, our target source language model was quite primitive and state-of-the art language on which aspects operate, such as Java, are more complicated. This complexity includes but is not limited to types & polymorphism, dynamic method invocation, and exceptions. We are working to extend our target language in order to reason about aspects operating on these.
- Reasoning about the join points of an AOP program is only half the story; the other half is reasoning about the advice placed on these points. We hope

to move into this area soon and begin developing analyses that will help reason about whole aspects in the absence of target programs.

## 7 Conclusions

We have presented two new problems in the domain of AOP: PCD satisfiability and call graph enumeration. We have shown both NP-complete, but as we show in the future work of Section 6, we are hopeful to find a restricted form of the problem that is polynomial. But why is this useful? We argued that call graph enumeration could be used to reason about aspects in the absence of target programs, and only after an understanding of this, can useful work be done to build component aspects. Tools built for AOP program understanding could use a call graph inference mechanism to help the programmer reason about the aspect she is writing. Testers would benefit from knowledge about the domain on which their aspects apply. This way they could have a feeling of the *edge-case* programs. Lastly, many AOP technologies come about with much thought of deep reasoning of them. This is our first step to understand aspects as sole entities not tied to target programs.

## 8 Acknowledgments

Many thanks to Theo Skotiniotis for his help.

## References

1. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming, Springer Verlag (1997) 220–242
2. Elrad, T., Aksit, M., Kiczales, G., Lieberherr, K., Ossher, H.: Discussing Aspects of AOP. *Communications of the ACM* **44** (2001) 33–38
3. Wand, M., Kiczales, G., Dutchyn, C.: A semantics for advice and dynamic join points in aspect-oriented programming. *TOPLAS* (2003) to appear. Earlier versions of this paper were presented at the 9th International Workshop on Foundations of Object-Oriented Languages, January 19, 2002, and at the Workshop on Foundations of Aspect-Oriented Languages (FOAL), April 22, 2002.
4. Filman, R.E., Friedman, D.P.: Aspect-Oriented Programming is Quantification and Obliviousness. In: Workshop on Advanced Separation of Concerns, OOPSLA, Minneapolis, USA (2000) <http://ic-www.arc.nasa.gov/ic/darwin/oif/leo/filman/text/oif/aop-is.pdf>.
5. Walker, D., Zdancewic, S., Ligatti, J.: A theory of aspects. In: Proceedings of the eighth ACM SIGPLAN international conference on Functional programming, ACM Press (2003) 127–139
6. Douence, R., Motelet, O., S&#252;dholt, M.: A formal definition of crosscuts. In: Proceedings of the Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Springer-Verlag (2001) 170–186

7. Tucker, D.B., Krishnamurthi, S.: Pointcuts and advice in higher-order languages. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 158–167
8. Palsberg, J.: Class-graph inference for adaptive programs. Theory and Practice of Object Systems, John Wiley and Sons, Inc. **3** (1997)
9. Alexander Koller, Joachim Niehren, R.T.: Dominance constraints: Algorithms and complexity. In: Third International Conference on Logical Aspects of Computational Linguistics (LACL '98), Grenoble, France (1998)
10. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.: An Overview of AspectJ. In Knudsen, J., ed.: European Conference on Object-Oriented Programming, Budapest, Springer Verlag (2001)
11. Gosling, J., Joy, B., Steele, G.L.: The Java Language Specification. Addison-Wesley Longman Publishing Co., Inc. (1996)
12. Lieberherr, K.J.: Adaptive Object-Oriented Software: The Demeter Method with Propagation Patterns. PWS Publishing Company, Boston (1996) 616 pages, ISBN 0-534-94602-X.
13. Lieberherr, K., Wand, M.: Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University (2001)
14. Wu, P., Lieberherr, K.: Compilation of Pointcut Designators using Traversals. Technical Report NU-CCIS-03-16, Northeastern University (2003)
15. Wu, P.: Supporting Proof (2003)
16. Bodirsky, M., Kutz, M.: Pure dominance constraints. (2002) 287
17. Dantas, D., Walker, D.: Aspects, information hiding and modularity. Technical report, Princeton University (2003)
18. Lieberherr, K., Lorenz, D.H., Ovlinger, J.: Aspectual collaborations: Combining modules and aspects. The Computer Journal **46** (2003) 542–565
19. McDirmid, S., Flatt, M., Hsieh, W.: Jiazzi: New-age components for old-fashioned Java. In: Proc. of OOPSLA. (2001)
20. McDirmid, S., Hsieh, W.C.: Aspect-oriented programming with jiazzi. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 70–79
21. Sereni, D., de Moor, O.: Static analysis of aspects. In: Proceedings of the 2nd international conference on Aspect-oriented software development, ACM Press (2003) 30–39
22. Masuhara, H., Kiczales, G., Dutchyn, C.: Compilation semantics of aspect-oriented programs (2002)
23. Althaus, E., Duchier, D., Koller, A., Mehlhorn, K., Niehren, J., Thiel, S.: An efficient graph algorithm for dominance constraints. Journal of Algorithms **48** (2003) 194–219 Special Issue of SODA 2001.
24. Reyle, U.: Dealing with ambiguities by underspecification: Construction, representation and deduction. Journal of Semantics **10** (1993) 123–179