Expressiveness and Complexity of Crosscut Languages

Karl J. Lieberherr
Northeastern University,
Boston, MA
lieber@ccs.neu.edu

Jeffrey Palm
Northeastern University,
Boston, MA
jpalm@ccs.neu.edu

Ravi Sundaram
Northeastern University,
Boston, MA
koods@ccs.neu.edu

ABSTRACT

Selector languages, or crosscut languages, play an important role in aspect-oriented programming (AOP). Examples of prominent selector languages include the pointcut language in AspectJ, traversal specifications in Demeter, XPath, and regular expressions. A selector language expression, also referred to as a *selector*, selects nodes on an instance graph (an execution tree or an object tree) that satisfies a meta graph (a call graph or a class graph). The implementation of selector languages requires practically efficient algorithms for problems such as: Does a selector always (or never) select certain nodes **Select-Always** (**Select-Never**), does a selector ever select a node **Select-Sat**, does one selector imply another selector **Select-Impl** or may an edge in an instance graph lead to a node selected by the selector **Select-Completion**.

We study these problems from the viewpoints of two important selector languages called SAJ, inspired by AspectJ, and SD, inspired by Demeter, and several of their sublanguages. We show a polynomial-time two-way reduction between SD and SAJ revealing interesting connections promoting transfer of algorithmic techniques from AspectJ to Demeter and vice-versa. We provide several practically useful polynomial-time algorithms for some of the problems, and we show others to be NP- or co-NP-complete. We present a fixed parameter tractable (FPT) algorithm for one of the NP-complete problems. This early result indicates a line of attack for dealing with the intractability inherent in these problems.

The paper provides a list of algorithmic results that are of interest to developers of scalable AOP tools. We discuss the consequences of this paper for our DAJ implementation.

General Terms

AspectJ, Demeter, pointcut designators, traversal strategies

1. INTRODUCTION

Aspect-oriented programs consist of two building blocks: WhereToInfluence and WhatToDo. The WhereToInfluence part defines the points in an executing program where we want to influence the program. The WhatToDo part defines how to influence the program. In this paper we analyze declarative, non Turing-complete selector (or crosscut) languages to formulate the WhereToInfluence part.

In a pioneering paper, Masuhara and Kiczales [16] compare crosscutting in four aspect-oriented mechanisms, including AspectJ and Demeter. We extend their work to include both algorithmic upper bounds as well as hardness results on several computational problems underlying AspectJ and Demeter. For example, motivated by another influential paper by Masuhara and Kiczales [17], we show that general elimination of run-time tests in AspectJ programs, even without negation in the pointcuts, is NP-complete in the general case.

Our analysis is at a high level of abstraction, yet detailed enough to provide useful practical input for the implementation of selector languages. The analysis is useful to current tools, e.g., AspectJ and Demeter (DemeterJ, DJ, DAJ[2]), and for many more aspect-oriented languages to come. Our model is a three level model [11] where at the top level we have selectors (e.g., pointcut designators or traversal strategies), at the second level meta graphs (e.g., static call graphs or class graphs) and at the third level instance trees (e.g., dynamic call trees or object trees) conforming to the meta graphs. The purpose of the selectors is to choose a set of nodes in the instance trees, or equivalently to choose a set of paths from the root of the trees to those nodes. For an example, the meta graph for the AspectJ program in Figure 2 is sketched in Figure 1.

We study several algorithmic problems for two kinds of selector languages and their sublanguages. The first language, called SAJ, is an abstraction of the AspectJ pointcut language. We lump all primitive pointcuts together into a term $n(l)$, selecting all the nodes with label l. We use `flow`(S), selecting all nodes reachable from the root through a node in S. And we add the set-theoretic operators `|`, `&` and `!`.

The second language, SD, is an abstraction and generalization of the Demeter traversal strategies. We use the version described in Palsberg *et al.* [21] but extended with the set-theoretic operators `&` and `!`. SD is more flow oriented, and we reuse the semantics from [21] in terms of path sets.

We consider two kinds of applications of selector languages.

AspectJ-style applications: The selector language is used to select nodes in the execution trees and their corresponding shadows in the program. The virtual machine decides, based on the input data, which execution tree to construct and the tree is traversed in full but only a subset of the nodes satisfies the selector expression. The term pointcut language is used instead of selector language.

Demeter-style applications: The selector language is used to select nodes in the object trees and their corresponding shadows in the meta graph. The object tree is given as input, and the tree is partially traversed reaching all the nodes satisfying the selector expression. The term traversal language is used instead of selector language.

One point of this paper is to also consider SAJ for Demeter-style applications and SD for AspectJ-style applications. The paper points out the close relationship between those two languages. We consider the following algorithmic problems for SAJ and SD and their sublanguages. For all of those problems we consider the version where the meta graph is given and for Select-Sat-Static we consider the case where only the selector is given as input and we ask for the existence of a suitable meta graph. **Select-Always:** Does a selector always select nodes with label A in all instances? This problem is useful for AspectJ-style applications of selector languages: it frees us from having to do any checking at run-time. See papers by Masuhara/Kiczales [17], Oege deMoor [23], and Wu/Lieberherr [27]. **Select-Always** is also useful for Demeter-style applications of selector languages: We are not required to do any run-time checking to ensure that the traversal is at the right place.

Select-Never: This is similar to the previous item. Does a selector select no nodes with label A in any instance?

Select-Sat-Static: Does a selector ever select a node? Here, we check whether a given selector has an effect on at least one instance graph by selecting at least one node. Selectors that never select a node are useless and should be corrected.

Select-Sat: Like **Select-Sat-Static**, except that in addition to the selector a meta graph is also given as input.

Select-Impl: Does one selector imply another selector? **Select-Impl** is useful in predicate dispatch languages, such as Fred [19] and Socrates [20], where inheritance is replaced by predicate implication. We cover here the special case where the predicates are declarative.

Select-First: Does an edge in an instance graph lead to a node selected by the selector? This is useful for guiding traversals [11] and for deciding whether a selector influences a particular branch of the execution of a program [17]. Our results in this paper should be considered in the context of the General Pointcut Satisfiability Problem:

> Given an AspectJ pointcut p and a Java program G,
> is there an execution of G in which p will select
> at least one join point?

This problem is undecidable even for a very simple pointcut language because the undecidability comes from the conditional statements in G.

Therefore we consider a conservative approximation of the program in the form of a call graph. We assume that all calls inside a procedure could happen. Because of the simple structure of meta graphs, we treat dynamic dispatch in a very simple way: zero or more of the calls could happen.

In this paper we show two kinds of results: lower-bound results, like NP-hardness and co-NP-completeness results and upper-bound results, like that certain checking problems can be solved in polynomial time. For the lower-bound results it is sufficient to consider only very limited programs, e.g., programs that only contain calls (without conditional statements). For the usefulness of our upper-bound results the conservative approximation mentioned above is an issue that needs to be explored further. The conservative approximation allows for many more possible program executions than can happen in practice. But still the upper-bound results are interesting because universally quantified statements (over all executions/instances) for the approximation are correct statements for the real program.

The following properties are preserved by the conservative approximation: not Select-Sat, Select-Always, Select-Never, Select-Impl, not Select-First. Note that Select-Sat is not preserved by the approximation because the meta graph might have an instance in which a join point is selected but that instance might never happen as an execution in the real program.

We show a polynomial-time two-way reduction from SD to SAJ revealing interesting connections and promoting the transfer of algorithmic techniques from AspectJ to Demeter and vice-versa. We provide several practically useful polynomial-time algorithms for some of the problems ,and we show others to be NP-complete or co-NP-complete. We present a fixed parameter tractable (FPT) algorithm for one of the NP-complete problems. This early result indicates a line of attack for dealing with the intractability inherent in these problems.

Our NP-completeness proofs are simple but not trivial. For example, we show that satisfiability and other problems for AspectJ pointcuts without complement are already NP-complete. The point of our reduction is that when we translate a boolean formula to a pointcut satisfiability problem, we can use the graph to simulate negation although the pointcut language does not itself contain negation.

In this paper we often refer to the traversal graph defined in [13, 11]. For the purpose of this paper we view the traversal graph as the Cartesian product of two graphs, where one graph is the meta graph and the other is the graph version of the SD selector expression. The Cartesian graph product $G = G_1 \times G_2$ of graphs G_1 and G_2 with disjoint point sets V_1 and V_2 and edge sets E_1 and E_2 is the graph with point set $u = (u_1, u_2)$ and $v = (v_1, v_2)$ adjacent with whenever $[u_1 = v_1$ and u_2 adj $v_2]$ or $[u_2 = v_2$ and u_1 adj $v_1]$ [8]. We note that the meta graph structure and selector language in [11] are more expressive and hence required a more elaborate

construction of traversal graphs.

Our paper uncovers novel aspects of the interplay between predicates and graphs. We believe that there is potential for further connections between this paper and the seminal work of Courcelle relating logic and graphs [1].

In summary the paper provides a novel framework for the study of the expressiveness of selector languages and their related algorithmic problems. We discuss the consequences of this paper for our DAJ implementation.

The rest of the paper is organized as follows: In section 2 we introduce our framework by defining meta graphs and instance graphs and our selector languages, SAJ and SD, including translations between them. In section 3 we introduce the problems, including the practical motivation behind them. Section 4 discusses a Fixed Parameter Tractable algorithm for Satisfiability with an application to Select-Sat. Section 5 contains related work and section 6 conclusions and future work.

2. GRAPH STRUCTURE AND SELECTOR LANGUAGE

For a particular graph there are a possibly infinite number of *instances* conforming to the graph structure, each of which, later, will be mapped to an AspectJ program execution call trace, or a Demeter object graph traversal. To select interesting points in an execution call trace or an object graph traversal, we have a general selector language which, later, will be mapped to AspectJ's pointcut designator language or Demeter's traversal specification.

2.1 Directed Graph and Instances

DEFINITION 1 (DIRECTED GRAPH). *A directed graph G is a pair $< V, E >$, where V is a set of vertices and $E \subseteq V \times V$ is a set of directed edges. There is a distinguished vertex $r \in V$, which is the starting vertex in G.* $\mathsf{Start}(G)$ *is defined on a graph G that returns its distinguished starting vertex for G from which all other nodes are reachable.*

We assume a labeling from nodes and edges to a finite alphabet, so that $\mathsf{Label}(x)$ is the label for a node or edge x.

DEFINITION 2 (INSTANCES OF GRAPH). *A directed graph I is called an instance of G, if I is a tree,* $\mathsf{Root}(I) = \mathsf{Start}(G)$ *and for each edge $e = (u, v) \in E(I)$, there is an edge $e' = (u', v') \in G$ so that* $\mathsf{Label}(u) = \mathsf{Label}(u')$ *and* $\mathsf{Label}(v) = \mathsf{Label}(v')$.

2.2 Paths

A *path* in a graph is a sequence $v_1 \dots v_n$ where $v_1, \dots, v_n$ are nodes of the graph; and $v_i \rightarrow v_{i+1}$ is an edge of the graph for all $i \in 1..n-1$. We call v_1 and v_n the source and the target of the path, respectively. If $p_1 = v_1 \cdots v_i$ and $p_2 = v_i \cdots v_n$, then we define the concatenation $p_1 p_2 = v_1 \cdots v_i \cdots v_n$.[1]

Suppose P_1 and P_2 are sets of paths where all paths in P_1 have the target v and where all paths of P_2 have the source v. Then we define[2]

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

$\mathsf{Paths}_\Phi(A, B)$ is defined as all paths from A to B in Φ where A and B are nodes of the meta graph Φ.

2.3 General Selector Language

We use two selector languages, SAJ and SD, based roughly on the selector languages of AspectJ and Demeter, respectively. SAJ has the form

$$S ::= l \mid \texttt{flow}(S) \mid S \texttt{ | } S \mid S \texttt{ \& } S \mid \texttt{!}S \qquad (1)$$

where l is a node label. The following are the evaluation rules for SAJ. We state them as a reduction, S_I:

$$\begin{aligned} S_I(l) &= \{v | v \in I \wedge \text{Label}(v) = l\} \\ S_I(\texttt{flow}(S)) &= \{v | \text{some } n \in S_I(S) \text{ reaches } v \in I\} \\ S_I(S_1 \texttt{ | } S_2) &= S_I(S_1) \cup S_I(S_2) \\ S_I(S_1 \texttt{ \& } S_2) &= S_I(S_1) \cap S_I(S_2) \\ S_I(\texttt{!}S) &= \backslash S_I(S) \end{aligned}$$

A *traversal specification* in SD has the form

$$D ::= [A, B] \mid D \cdot D \mid D \texttt{ | } D \mid D \texttt{ \& } D \mid \texttt{!}D \qquad (2)$$

where A and B are nodes of a meta graph. Such a specification denotes a set of paths in a given meta graph Φ, intuitively as follows:

Selector	**Set of paths**
$[A, B]$	The set of paths from A to B in Φ
$D_1 \cdot D_2$	Concatenation of sets of paths
D_1	D_2	Union of sets of paths
D_1 & D_2	Intersection of sets of paths
$!D$	All paths from $\mathsf{Source}(D)$ to $\mathsf{Target}(D)$ not satisfying D

For a traversal specification to be meaningful, it has to be well-formed. Formally, well-formedeness is defined in terms of two functions, Source and Target, which both map a specification to a node. The following chart shows the denitions for Source and Target where $\mathsf{Source}(D)$ is the source node determined by D, and $\mathsf{Target}(D)$ is the target node determined by D:

Selector: D	$\mathsf{Source}(D)$	$\mathsf{Target}(D)$
$[A, B]$	A	B
$D_1 \cdot D_2$	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_2)$
D_1 | D_2	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_1)$
D_1 & D_2	$\mathsf{Source}(D_1)$	$\mathsf{Target}(D_1)$
$!D$	$\mathsf{Source}(D)$	$\mathsf{Target}(D)$

A traversal specication is well-formed if it determines a source node and a target node, if each concatenation has a meeting point, and if each union of a set of paths preserves the source

[1] The v_i in a path don't have to be distinct. v_1 is a path from source v_1 to target v_1 where $n = 1$.

[2] $P_1 \cup P_2$ is the set union of the paths in P_1 and P_2.

and the target. This is expressed by the predicate WF:

$$\begin{aligned}
\mathsf{WF}([A,B]) &= \text{true}\\
\mathsf{WF}(D_1 \cdot D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\quad \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2)\\
\mathsf{WF}(D_1 \mid D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\quad \mathsf{Source}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2) \wedge \\
&\quad \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Target}(D_2)\\
\mathsf{WF}(D_1 \,\&\, D_2) &= \mathsf{WF}(D_1) \wedge \mathsf{WF}(D_2) \wedge \\
&\quad \mathsf{Source}(D_1) =_{\mathsf{nodes}} \mathsf{Source}(D_2) \wedge \\
&\quad \mathsf{Target}(D_1) =_{\mathsf{nodes}} \mathsf{Target}(D_2)\\
\mathsf{WF}(!D) &= \mathsf{WF}(D)
\end{aligned}$$

If D is well-formed and compatible with Φ, then $\mathsf{PathSet}_\Phi(D)$ is a set of paths in Φ from the source of D to the target of D, defined as follows:

$$\begin{aligned}
\mathsf{PathSet}_\Phi([A,B]) &= \mathsf{Paths}_\Phi(A,B)\\
\mathsf{PathSet}_\Phi(D_1 \cdot D_2) &= \mathsf{PathSet}_\Phi(D_1) \cdot \mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(D_1 \mid D_2) &= \mathsf{PathSet}_\Phi(D_1) \cup \mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(D_1 \,\&\, D_2) &= \mathsf{PathSet}_\Phi(D_1) \cap \mathsf{PathSet}_\Phi(D_2)\\
\mathsf{PathSet}_\Phi(!D) &= \mathsf{Paths}_\Phi(\mathsf{Source}(D), \mathsf{Target}(D))\\
&\quad - \mathsf{PathSet}_\Phi(D)
\end{aligned}$$

We show a reduction from SD to SAJ. In the following, SD expressions are on the left-hand side and SAJ expressions are on the right:

$$\begin{aligned}
T([A,B]) &\rightarrow \texttt{flow}(A)\ \texttt{\&}\ B\\
T(D_1 \cdot D_2) &\rightarrow \texttt{flow}(T(D_1))\ \texttt{\&}\ T(D_2)\\
T(D_1 \mid D_2) &\rightarrow T(D_1) \mid T(D_2)\\
T(D_1 \,\&\, D_2) &\rightarrow T(D_1)\ \texttt{\&}\ T(D_2)\\
T(!D) &\rightarrow !T(D)
\end{aligned}$$

Here is an example reduction of $[A,B] \cdot [B,C]$:

$$\begin{aligned}
T([A,B] \cdot [B,C]) &\rightarrow \texttt{flow}(\texttt{flow}(A)\ \texttt{\&}\ B)\ \texttt{\&}\ \texttt{flow}(B)\ \texttt{\&}\ C\\
&= \texttt{flow}(\texttt{flow}(A)\ \texttt{\&}\ B)\ \texttt{\&}\ C\\
&= \texttt{flow}(A)\ \texttt{\&}\ \texttt{flow}(B)\ \texttt{\&}\ C
\end{aligned}$$

We also show an informal[3] reduction from SAJ to an SD expression D. In the following, SAJ expressions are on the left-hand side, and SD expressions are on the right:

$$\begin{aligned}
T(n(l)) &\rightarrow [\mathsf{Source}(D), l]\\
T(\texttt{flow}(l)) &\rightarrow [\mathsf{Source}(D), l] \cdot [l, \mathsf{Target}(D)]\\
T(S_1 \mid S_2) &\rightarrow T(S_1) \mid T(S_2)\\
T(S_1\ \texttt{\&}\ S_2) &\rightarrow T(S_1)\ \texttt{\&}\ T(S_2)\\
T(!S) &\rightarrow !T(S)
\end{aligned}$$

3. PROBLEMS

In the following section we present various problems related to selector expressions and reason about their complexity. Theorems are presented in tables of the form:

[3]This is informal because a resultant in SD could have multiple targets.

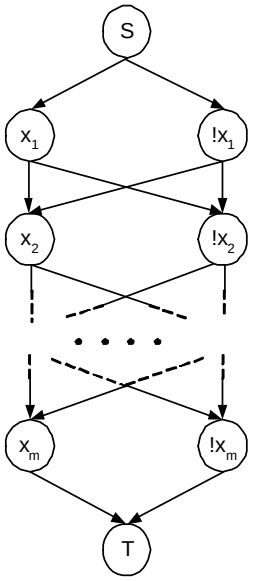

Figure 1: Ladder graph.

	SD	SAJ
-	R_1	R_2
&	R_3	R_4
!	R_5	R_6

Each R_i is a complexity result. The first row represents complexity results for the languages shown in grammars (1) and (2) without intersection or negation, called the base language; the second row shows results for these languages without negation; and the third row shows results for these languages without intersection. A Y in a result represents a problem that is trivially true. All proofs are in [12].

We split this section according to general problems – e.g. Select-Sat. We refer to particular instances of these problems for certain languages by the form $A/B/C$ where A is a general problem or $*$ for all problems, C is the language SD or SAJ, B is one of $-$, &, or ! representing the version of language C. For example, Select-Sat/&/SAJ represents the Select-Sat problem over the SAJ language with intersection, and */-/SD represents any problem on the base language, -, over the SD language.

We use *ladder graph*, as shown in Figure 1, as our main tool to represent boolean forumulas. This graph consists of a root s, target t, and nodes x_i and $!x_i$ for $i = 1$ to m. A path from s to t must path through only one x_i or $!x_i$ for all i to reflect the fact that each literal in a boolean formula may be assigned either true or false; but not both. In addition, we use the following generic constructions.

Many of the problems have similar complexity results, which are given in Table 3.

3.0.1 SD Generic Construction

For the */-/SD case, we turn the selector into a graph p' ($[A,B]$ becomes an edge from A to B.) Then we construct the cross product traversal graph $T(G,p')$ [13, 11].

Problem	SD	SAJ
-	P	P
`&`	NP-complete	NP-complete
`!`	NP-complete	NP-complete

Table 1: Complexity results for many problems.

The motivation for the cross product $T(G, p')$ is as follows: Implementing the strategy $S = [A, B]$ on a class graph G [14] is straight-forward (called the FROM-TO computation): In G we do a forward depth-first traversal from A and a backward depth-first traversal from B and take the intersection of the two. The resulting graph succinctly represents the desired path set. For a general strategy we want to reduce the problem of succinctly representing the path set to the FROM-TO problem and this reduction is achieved by replacing the class graph with a much larger graph and doing the FROM-TO computation in that graph. This much larger graph is precisely the cross product of the strategy and the class graph.

3.0.2 SAJ Generic Construction

We need a generic construction for the */-/SAJ case. We use the */-/SD case as a guide. In the */-/SD case we flag each edge selected by a primitive `flow`$(A \cdot B)$ with $A \cdot B$. This is basically the idea behind the traversal graph construction. We need this labeling to avoid information loss (i.e. the short-cuts and zigzags of Palsberg *et al.*, [21]). We use a similar approach for */-/SAJ. The edges selected by each primitive `flow`(A) are labeled by `flow`(A). We can reduce the SAJ expression to the form s_1 `|` $\cdots$ `|` s_k for $1 \leq k$, where each s_i is in the form of either $n(l)$ or `flow`(s') because

$$\texttt{flow}(n(A_1) \texttt{ | flow}(n(A_2))) = \texttt{flow}(n(A_1)) \texttt{ | flow}(n(A2)).$$

Therefore we can build in polynomial time a structure, called the flow graph, that plays the same role as the traversal graph. The size of the flow graph is bounded by the size of the meta graph times the number of flow expressions in the selector (after removal of nested flows).

We use this construction for */-/SAJ where * in Select-Never (is the node ever in the flow graph?), Select-Sat (is the flow graph empty?), Select-Impl (is one flow graph a subgraph of another flow graph?) and Select-First (which edges are in the flow graph?).

In our NP-completness proofs we leave out the part that shows that a problem is in NP and we focus on the harder NP-hard part. We leave the NP membership part as an exercise to the reader.

3.1 Select-Sat

We are presenting a proof sketch of one of our complexity-theoretic results as an example of the kinds of gadgets we use in our reductions.

Definition 3 (Select-Sat). *Given a selector p and a meta graph G, is there an instance tree for G for which p selects a non-empty set of nodes.*

SAJ Expression			Pointcut
p_1	=	$x1$ `|` $!x2$ `|` $x3$	`p1()`
p_2	=	$!x1$ `|` $x2$	`p2()`
p_3	=	$x1$	`p3()`
p_4	=	$!x3$	`p4()`
p_{all}	=	p_1 `&` p_2 `&` p_3 `&` p_4	`all()`

Table 2: SAJ expressions and AspectJ pointcuts.

Table 3 shows the complexity results for Select-Sat. The Select-Sat/-/SD problem has been implemented for a special case in Demeter/C++ and for the general case in DemeterJ, DJ and DAJ. Our users demanded such a test because knowing that a traversal specification (selector) will never select a node indicates, usually, a false assumption about the class graph (meta graph). Select-Sat/*/SAJ is not currently implemented in AspectJ, and this can make it harder to debug pointcut designators. A small typo in one of the pointcuts may empty the set of selected join points. It would be helpful to get a warning for the pointcuts that select an empty set of join points. We hope that our FPT algorithms in Section 4 will lead to interesting algorithms for the NP-complete cases for AspectJ and for Demeter.

Proof *Select-Sat/&/SD.* The proof is by reduction from 3-SAT. Consider a 3-SAT formula ϕ. Let $v_1, v_2, \ldots, v_n$ be the variables. Create a meta graph that is a dag as follows: a source s with arcs going to x_1 and $!x_1$, arcs from x_i and $!x_i$ to x_{i+1} and $!x_{i+1}$ and finally from x_n and $!x_n$ to a sink t. This is $G(\phi)$, called a ladder graph, as shown in Figure 1. Now create an atomic selector for each literal and create the total selector $S(\phi)$ by taking the union and intersection over literals for each clause. For a literal $li = v_i/!v_i$ create the selector "from s to t via li" – i.e. "$[s, v_i] \cdot [v_i, t]$". Clearly, $(S(\phi), G(\phi))$ is satisfiable iff ϕ is satisfiable. □

Our reduction constructs a meta graph and a selector from the Boolean formula. But our meta graph is really an abstraction of a Java program and the selector an abstraction of an AspectJ pointcut designator. An important point of our paper is that the meta graph/selector abstraction is good enough to reason about the computational complexity at the AspectJ level. To demonstrate this point, we translate an example boolean formula shown in Table 2 directly to an AspectJ pointcut in Figure 2. Here, `x1`, `x2`, `x3`, `nx1`, `nx2`, and `nx3` in Figure 2 correspond to x_1, x_2, x_3, $!x_1$, $!x_2$, and $!x_3$, respectively.

3.2 Select-Sat-Static

Definition 4 (Select-Sat-Static). *Given a selector p, is there a meta graph G and an instance tree for G for which p selects a non-empty set of nodes.*

A Select-Sat-Static test is a must for a "perfect" aspect-oriented system, because a selector that fails for all meta graphs is clearly useless. Yet, both AspectJ and the Demeter Tools don't implement such a test, maybe, because it is perceived to be unlikely that a user writes such pointcuts or traversal strategies. Again, we hope that our FPT ideas in Section 4 will help to develop practically useful algorithms.

```
public class Example {
 public static void main(String[] s) {x1(); nx1();}
 static void  x1() { x2(); nx2(); }
 static void  x2() { x3(); nx3(); }
 static void  x3() {    target(); }
 static void nx1() { x2(); nx2(); }
 static void nx2() { x3(); nx3(); }
 static void nx3() {    target(); }
 static void target() {}
}
aspect Aspect {
 pointcut p1():  cflow(call (void  x1()))
   ||            cflow(call (void nx2()))
   ||            cflow(call (void  x3()));
 pointcut p2() : cflow(call (void nx1()))
   ||            cflow(call (void  x2()));
 pointcut p3() : cflow(call (void  x1()));
 pointcut p4() : cflow(call (void nx3()));
 pointcut all(): p1() && p2() && p3() && p4();
 before(): all() && !within(Aspect) {
   System.out.println(thisJoinPoint);
 }
}
```

Figure 2: AspectJ example.

The following are the complexities for Select-Sat-Static:

Select-Sat-Static	SD	SAJ
-	Y	Y
&	Y	Y
!	NP-complete	NP-complete

We mention also that the following problem is NP-complete for both SAJ and DJ (even without complement) if we allow that an instance may be a directed acyclic graph (dag), not just a tree. Since a tree is a dag, we restrict our definition of the problem to trees.

Definition 5 (Select-Sat-Dynamic). *Given a selector p, a meta graph G, and an instance tree I for G, does p select a non-empty set of nodes in I?*

3.3 Select-Impl

Definition 6 (SEL). $\mathsf{SEL}(s, G, I)$ *is the set of nodes selected by s in I (which conforms to G).*

Definition 7 (Select-Impl). *Given two selector expressions s_1 and s_2 and a graph G, for all instances I of G: SEL(s_1,G,I) is a subset of SEL(s_2,G,I).*

Predicate-dispatch-based aspect languages such as Socrates [20] use selector implication as a primitive to generalize inheritance. Selector implication is also useful in other applications. For example, a security policy might state that a set of nodes accessible by one role (e.g., worker) must always be a subset of the set of nodes accessible by another role (e.g., manager). Table 3 shows the complexity results for !Select-Impl– hence in this table all NP-complete results are co-NP-complete results for Select-Impl.

3.4 Select-First

Definition 8 (Select-First). *Given a selector p, a meta graph G, and an instance I, compute the set of outgoing edges from a node of I satisfying G that might lead to a target node selected by p.*

In the Demeter case the Select-First predicate is the fundamental tool to implement traversals efficiently. The approach is to combine the selector and meta graph into a new graph that for each node tells which outgoing edges are worthwhile traversing. Worthwhile means that it may lead to a target node satisfying p in an appropriate subobject. See [15] for the generalization of this predicate to class graphs with is-a and has-a edges. [21] contains an efficient implementation for a special case that was used in Demeter/C++. The D*J tools use the AP Library [13] that implements Select-First/-/SD using the ideas in [11].

The NP-completeness result for Select-Sat/&/SD has interesting implications for the semantics of traversals as we make the selector language more expressive. The DAJ tool [2] is an extension of AspectJ with traversals and strategies. Using the AspectJ declare construct we could write:

```
declare strategy: sname: "{A -> B}" ;
declare traversal: void foo(): sname(Visitor);
```

In this DAJ example the expression `"A -> B"` is analogous to the SD expression $[A, B]$ This selector expression uses SD without negation but with intersection. This traversal defines an adaptive method called `foo` using the strategy named `sname` and the `Visitor`, which is a normal Java class. In DAJ intersection is used frequently because it also plays the role of cleaning the class graph from unwanted information.

The semantics of a traversal is defined in terms of Select-First [11, 15]. This works well for SD without intersection and complement because we have an efficient algorithm. In the presence of intersection, we currently implement the following solution: We assume that intersection only appears at the outermost level. This is a reasonable assumption. To implement (s_1 `&` s_2), compute the traversal graph t_1 for s_1 and G and the traversal graph t_2 for s_2 and G. Then we simulate both t_1 and t_2 on an instance graph. But unfortunately this gives the wrong semantics because we might go down an edge in the instance graph although it never leads to a target. Instead we need to construct the cross product of t_1 and t_2, leading to an explosion in the number of nodes if we do this multiple times. We know now that there is no way around this because of the NP-completeness of the underlying problem.

For the AspectJ case the predicate is useful to implement `cflow`. It tells us along which execution paths we are in the scope of a pointcut designator where we have to execute advice. Table 3 shows the complexity results for Select-First.

Consider a selector expression p and a meta graph G in Select-Sat/&/SAJ. Let's assume that we can compile p and

G into a function Super(r) that given a node r of an instance conforming to G, computes the set of outgoing edges from r that may lead to a selected node. The function Super encodes the information about p and G into a form that is useful for deciding which edges are worthwhile to traverse to reach a target node.

Let's assume that we can construct Super in polynomial-time and that Super runs in polynomial-time. This would create a polynomial algorithm for Select-Sat/&/SAJ. Namely, we compile the pair (p, G) into Super(r) and run Super(r) on an instance I of G that has the root and an edge to each of the successors of the root. Note that for each meta graph G we can generically construct such an instance. Clearly, the size of r is bounded by the size of G. The input (p, G) is satisfiable iff Super(r) returns a non-empty set on I; i.e., there is an instance graph in which at least one node is selected.

Note that, the same argument holds for: Select-Sat/&/SD. In order to prove that (p, G) is unsatisfiable (co-NP-complete problem) we need only run Super on a generically constructed instance.

As soon as the selector language becomes too powerful, selecting nodes in instances becomes expensive. We can use this to prove that Select-First/&/SAJ and Select-First/&/SD are NP-complete.

3.5 Select-Always

DEFINITION 9 (SELECT-ALWAYS). *Given a selector p and a meta graph G and a node n in G, for all instance graphs I of G all of the instances of n in I are selected by p.*

If an AspectJ or Demeter compiler could answer this question efficiently we could drastically speed up compilation time. Table 3 shows the complexity results for !Select-Always.

3.6 Select-Never

DEFINITION 10 (SELECT-NEVER). *Given a selector p and a meta graph G and a node n in G, for all instance graphs I of G none of the instances of n in I are selected by p.*

In addition to the benefits found from Select-Always, efficient solutions to this problem could provide useful feedback to users when writing pointcuts or traversals. Often one writes a pointcut and then refactors a system. The user would want to know when her pointcuts were possibly no longer valid after this refactoring. This is just one example of why this is an important problem. Table 3 shows the complexity results for !Select-Never.

4. FPT ALGORITHMS

We have shown that Select-Sat is NP-complete. As noted in [6] the fact that a problem has been shown to be NP-hard is not a cause for despair. All it really means is that the initial hope for an exact general algorithm is in vain. There are a few different avenues of attack at this point - the use of randomness, the search for good approximate solutions and use of parametrization. Here we focus on this last approach.

We look more closely at the structure of the input. Select-SAT consists of a meta graph and a selector. We have shown this problem to be NP-hard even when the meta graph is the ladder graph and the selector is a 3-SAT formula. In practice though, it is often the case that the selector rarely has too many clauses. In particular we consider situations where our meta graph is a generalization of the ladder graph and the conjunctive selector formula has only k clauses. We ask the question - what is the behavior for a fixed k? Observe that the naive approach of trying every possible setting of the variables in the selector leads to an exponential-time (2^n) algorithm. We now demonstrate that in fact for fixed k, this problem, which we call the k-generalized-ladder-Select-Sat, is solvable in time that is linear in the size of the formula and the graph.

The approach of parametrization has been developed by Downey and Fellows in a seminal series of papers [3]. They show that the usual combinatorial explosion involved in NP-hard problems can often be handled if one can get one's hands on the right parametrization. In cases where such a parametrization exists, the problem is said to be *Fixed Parameter Tractable*. More precisely, a parametrized problem $< x, k >$, where x is the input and k the parameter, is said to be in FPT if there exists an algorithm and a constant c (independent of k), and a function f such that the algorithm accepts valid inputs in time $f(k)|x|^c$. Note for example that Vertex Cover is in FPT where k, the size of the cover, is fixed. On the other hand Independent Set with k representing the size of the independent set continues to be intractable even when k is fixed.

We now define the problem k-generalized-ladder-Select-Sat and present a fast kernelization scheme to solve it.

DEFINITION 11. *k-generalized-ladder-Select-Sat consists of a generalized ladder graph and a selector formula in conjunctive normal form. The generalized ladder graph is a directed acyclic leveled graph that has a unique source s and unique sink t. The graph contains all edges between adjacent levels. At each level the graph has no more than $f_i(k)$ vertices, where i represents the level. See Figure 3. The selector formula is in CNF and has at most k clauses.*

Note that our earlier NP-hardness proof goes through for k-generalized-ladder-Select-SAT when k is considered to vary with n, instead of being fixed.

THEOREM 1. *k-generalized-ladder-Select-Sat is in FPT.*

PROOF. At a high level our strategy is to find in time polynomial in n, a kernel or the hard core of the problem which only depends on k and not on n; and then we employ a search tree strategy to try all possible cases in the kernel. Let $f_{\max} = \max_i f_i(k)$ denote the maximum number of vertices over all rows of the gneralized ladder graph.

Kernelization. Consider the selector formula. Each literal is of the form v where v is a vertex in the asociated generalized ladder graph and selects the set of paths from s to t going

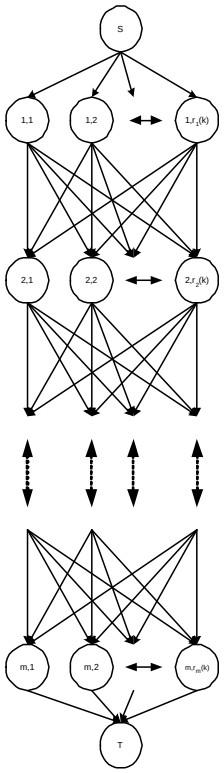

Figure 3: General ladder graph.

through that vertex v. If the formula has any single literal clauses then since all paths from s to t satisfying the formula must pass through that vertex we can prune the metagraph by removing all vertices other than v from its level. Note that in this manner we account for all single literal clauses or the metagraph gets pruned into the empty graph in which case we know that the selector formula is unsatisfiable. We are now left to consider the case where we have taken care of all single literal clauses, i.e. we can assume that the formula only consists of clauses with 2 or more literals. Consider any clause with more than $k * f_{\max}$ literals. Observe, that to satisfy each of the remaining (upto) k clauses we need to only satisfy 1 literal in each clause. Since the clause in consideration has more than $k * f_{\max}$ literals that means this clause contains a literal that is on a level of the meta graph different from that of any other vertex needed for satisfying any of the other clauses. Hence such a clause can be trivially satisfied. Thus we can eliminate all clauses with more than $k * f_{\max}$ literals. Thus we are left with a formula with at most k clauses where each clause has between 2 and $k * f_{\max}$ literals.

Search tree. Now try setting to true all possible choices of literals, one from each clause, there are at most $k^{k*f_{\max}}$ possible choices and for each possible choice compute the subgraph of the meta graph that satisfies that choice. If all subgraphs are empty then we know that the selector is unsatisfiable. If some subgraph is nonempty then consider the clauses that were pruned for having more than $k * f_{\max}$ literals and pick a literal in each of these clauses on a level different from all the previously chosen literals and prune this subgraph so as to satisfy these clauses.

It is easy to see that the above scheme has running time $O(n) + O(k^{k*f_{\max}})$ and hence k-generalized-ladder-Select-Sat is in FPT. □

5. RELATED WORK

[16] is an interesting study of crosscutting mechanisms. They discuss both the WhereToInfluence-part and the WhatToDo-part while we focus on the WhereToInfluence-part only. But in their Table 1 they also put pointcuts and traversal specifications at the same level as we do in this paper. (Demeter actually uses another incarnation of AOP which is not discussed in either paper: The visitor signatures are pointcuts and the visitor method bodies are the advice.) The crosscut definition in [16] can be applied to selector languages: Two selectors p_1 and p_2 crosscut if the set of selected nodes intersect at the instance level or meta graph level but none is a subset of the other. Crosscutting of selector expressions is very typical especially if we consider the nodes along the paths as well (not just the target nodes).

The two papers differ in that we focus on algorithms and complexity results of selector languages.

In [17], the issue of unnecessary run-time checks in AspectJ is discussed. The meta graph is considered to be included in the program text. They use partial evaluation to remove unnecessary pointcut tests. They don't analyze the complexity of the underlying task but instead use a powerful, but potentially expensive tool, to attack the problem. We show that general elimination of run-time tests (Select-Never and Select-Always) is NP-complete in the general case.

In Eichberg et al. [5] they use functional queries as their selector language. This is an interesting generalization of the kind of selector languages discussed in this paper. It would be useful to analyze the combinatorial problems discussed in this paper for a simple functional query language as selector language. Eichberg et al. use XQuery (based on XPath) as the query language which supports the descendent axis (denoted by "`//`") that can express traversal like $[A, B]$ (from A to B) in our SD selector language.

The study of selector languages is an active topic in the database community over the past few years. Schwentick [22] does an extensive study of the equivalent of the Select-Impl problem for XPath and show it to be co-NP-complete for a particular subset of XPath. In a paper by Neven and Schwentick it is shown: Theorem 7. Containment of XP(DTD, /, //, *)-expressions is in P. This problem matches with our Select-Impl/-/SD which we also have shown to be in P [13]. DTD's correspond to our meta graphs. The difference with our work is that XPath slices the selector language world in a way that is different from AspectJ pointcuts (SAJ) or Demeter traversals (SD). Our paper also differs in that we provide a unifying model to study key properties of a wide variety of selector languages.

Sereni and de Moor [23] study the static determination of `cflow` pointcuts in AspectJ. They reason also in terms of sets of paths, but they use a regular expression style selector language. They model pointcut designators as automata which is similar to our translation of selectors into graphs.

They do whole program analysis on the program's call graph and try to determine whether a potential join point fits into one of the following three cases: (1) it *always* matches a `cflow` pointcut; (2) it *never* matches a `cflow` pointcut; (3) it *maybe* matches a `cflow` pointcut. In case (3), there is still a need to have dynamic matching code. They didn't analyze the computational complexity of (1, Select-Always) and (2, Select-Never). Our NP-completeness results for Select-Always and Select-Never complement their practical analysis.

In [4] an AspectJ compiler, called abc, is discussed and they found several improvements to implementing `cflow` over the AspectJ compiler ajc. Our work assumes a whole program analysis but should provide useful input to compiler writers. Using traversal graphs for compiling certain AspectJ programs should lead to even more speed-ups.

Mendelzon and Wood [18] analyzed the complexity of finding regular paths in graphs, which is similiar to our Select-First and Select-Sat problems with subtle differences. They showed that finding simple regular paths in a graph is NP-complete problem while finding regular paths is a polynomial-time problem (if the regular expression language is not too rich). Their selector language is a regular expression language that could be studied in a similar way we have sudied SAJ and SD. Mendelzon and Wood don't consider instance graphs: they operate at the level of selectors (regular expressions) and meta graphs only.

The work on JAsCo [24, 25] is using a pointcut-style notation and Demeter-style traversal specifications in the same system. The selector language approach described in this paper might lead to a tighter integration of the two languages.

Gybels and Brichau [7] present a number of language features that could be useful for expressing more expressive pattern-based crosscuts. The language presented is pattern-based, similar to that found in AspectJ [9], uses Prolog, and is implemented on SmallTalk. It first the adds unification as a feature, which allows variable binding. Another feature are object reifying predicates that (1) provide access to the "context object" property of the matched join point, (2) provide direct access to the state of objects, and (3) can express the way a certain object should respond to messages.

Lastly, join point shadows are used to access static properties of the program, and recursion is allowed in defintions. The latter makes this language Turing complete.

Walker presents the concept of *Implicit Context* in his dissertation [26]. Implicit context consists of three concepts: boundaries between conflicting world views, contextual dispatch which is used to alter communications, and communication history which is used to retrieve previous state when performing contextual dispatch. This allows a programmer to express the essential structure of our software modules, through the use of implicit context, to make those modules easier to reuse and the systems containing those modules easier to evolve. Expressing these context requires expressive languages which could benefit from our work.

Several papers use regular expressions as selector language [23] and [10]. Several of our results should carry over to regular expressions but the details need to be worked out in future work.

6. CONCLUSIONS AND FUTURE WORK

We have studied graph-theoretic decision problems fundamental to aspect-oriented software development. We have simplified our model by considering only meta graphs and instance graphs with has-a edges. But it is not hard to generalize our algorithms and proofs to more general meta graphs as has been done in [11] for Select-First/-/SD.

The simplified model promotes a succinct description of both upper and lower bounds for a variety of relevant problems. In doing so we have made contributions to complexity theory – a new FPT algorithm for a subset of Select-Sat and new NP-completeness proofs – and PL theory – two models of selector languages and a collection of related algorithms useful in AOSD tools (compilers, IDEs) that assume the whole world assumption.

The NP-completeness results are useful for three reasons: (1) The NP-completeness of the monotone version of the Satisfiability problem for the AspectJ pointcut language (Select-Sat/&/SAJ) is surprising because Satisfiability for monotone boolean formulas can be solved in polynomial-time.(2). They help us to steer around language features that might be expensive to implement. (3) In case we need the NP-complete language features, we can think carefully about what kind of algorithms degrade gracefully if certain features of the input are bounded. This is the topic of FPT.

Many of the efficient algorithms we describe are practically useful, and have not been described in the literature so far. We have implemented algorithms for several of the */-/SD problems in D*J and they are distributed separately through the AP Library. Select-First/-/SD is used heavily in the D*J tools whenever an object is traversed. An empirical study of traversals is in [28].

This is just the beginning in reasoning about the relationship between different pointcut languages and learning how to utilize different languages' features in an efficient manner. For example, a common AspectJ idiom is to capture a call only in certain contexts; say a call to `f()` but not underneath a call to `g()`. This is written in AspectJ as

```
call(void f()) & !cflow(void g())
```

We can use our results from this paper to see that reasoning about this statement uses an NP-complete sublanguage. However, we can write an equivalent Demeter traversal as

```
from main() bypassing g() to f()
```

that uses a polynomial-time sublanguage. So, we will use this framework to unify multiple pointcut languages in an intelligent manner.

In future work we want to study incremental versions of the problems which are important for incremental compilation. We also want to focus on studying *shy* selector languages. Both SAJ and SD are shy selector languages but they can be improved and maybe integrated. A "control-flow-shy"

selector language is discussed in [5]. In addition to minimizing information from the class graph, we want to minimize information from the control-flow graph in the selectors.

7. REFERENCES

[1] B. Courcelle. Graph rewriting: An algebraic and logical approach. In J. van Leeuwen et al, editor, *Handbook of Theoretical computer Science, Vol B.* North Holland, 1990.

[2] Doug Orleans and Karl J. Lieberherr. DAJ: Demeter in AspectJ home page. http://www.ccs.neu.edu/research/demeter/DAJ/.

[3] R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Springer, 1999.

[4] B. Dufour, C. Goard, L. Hendren, C. V. erbrugge, O. de Moor, and G. Sittampalam. Measuring the dynamic behaviour of aspectj programs. In D. Schmidt, editor, *OOPSLA*, Vancouver, CA, 2004. ACM Press.

[5] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In *The Second ASIAN Symposium on Programming Languages and Systems ASPLAS*, 2004.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1990.

[7] K. Gybels and J. Brichau. Arranging language features for more robust pattern-based crosscuts. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 60–69. ACM Press, 2003.

[8] F. Harary. *Graph Theory*. Addison Wesley, 1994.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. An Overview of AspectJ. In J. Knudsen, editor, *ECOOP*, Budapest, 2001. Springer Verlag.

[10] S. Krisnamurthi, K. Fisler, and M. Greenberg. Verifying aspect advice modularly. In *FSE*, 2004.

[11] K. Lieberherr, B. Patt-Shamir, and D. Orleans. Traversals of object structures: Specification and efficient implementation. *TOPLAS*, 26(2):370–412, 2004.

[12] K. J. Lieberherr, J. Palm, and R. Sundaram. Expressiveness and complexity of crosscut languages. Technical Report NU-CCIS-04-10, Northeastern University, September 2004.

[13] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997.

[14] K. J. Lieberherr and B. Patt-Shamir. Traversals of Object Structures: Specification and Efficient Implementation. Technical Report NU-CCS-97-15, College of Computer Science, Northeastern University, Boston, MA, Sep. 1997.

[15] K. J. Lieberherr and M. Wand. Traversal semantics in object graphs. Technical Report NU-CCS-2001-05, Northeastern University, May 2001.

[16] H. Masuhara and G. Kiczales. Modeling Crosscutting in Aspect-Oriented Mechanisms. In *ECOOP*, June 2003.

[17] H. Masuhara, G. Kiczales, and C. Dutchyn. Compilation semantics of aspect-oriented programs. In R. Cytron and G. Leavens, editors, *FOAL*, Enschede, Netherlands, 2002.

[18] A. O. Mendelzon and P. T. Wood. Finding regular simple paths in graph databases. In *VLDB*, 1989.

[19] D. Orleans. Incremental programming with extensible decisions. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede, The Netherlands, April 2002.

[20] D. Orleans. The Socrates Programming Language, September 2004. http://socrates-lang.sf.net/.

[21] J. Palsberg, C. Xiao, and K. J. Lieberherr. Efficient implementation of adaptive software. *TOPLAS*, 17(2):264–292, Mar. 1995.

[22] T. Schwentick. Xpath query containment. *SIGMOD Rec.*, 33(1):101–109, 2004.

[23] D. Sereni and O. de Moor. Static analysis of aspects. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 30–39. ACM Press, 2003.

[24] D. Suvée, W. Vanderperren, and V. Jonckers. JAsCo: an aspect-oriented approach tailored for component based software development. In *Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 21–29. ACM Press, 2003.

[25] W. Vanderperren. *Combining Aspect-Oriented and Component-Based Software Engineering*. PhD thesis, Vrije Universiteit Brussel, 2004.

[26] R. Walker. Essential software structure through implicit context. *Ph.D. dissertation, The University of British Columbia*, 2003.

[27] P. Wu and K. J. Lieberherr. Compilation of Pointcut Designators using Traversals. Technical Report NU-CCIS-03-16, Northeastern University, December 2003.

[28] P. Wu and M. Wand. An Empirical Study of the Demeter System. In *Proceedings of the SPLAT workshop of the 3rd international conference on Aspect-Oriented Software Development*, 2004.