Intersection and Negation of strategies
Intersection

• Intersect(S1,S2) for G:
  – Product graph of TG(S1,G) and TG(S2,G)
  – Need to add a graph intersection function to AP Library
Strategy S1:
\{A \rightarrow X \\
X \rightarrow C\}

\text{TG(S1,G)}

class graph

\begin{itemize}
\item A
\item B
\item C
\end{itemize}

start set

\begin{itemize}
\item X
\end{itemize}

finish set

thick edges with incident nodes: traversal graph
Strategy $S_2$:

$\{A \rightarrow B, B \rightarrow C\}$

TG($S_2, G$)

class graph

start set

finish set

thick edges with incident nodes: traversal graph
Edges in intersection
(1Ac1, 2Ac1) x> (1Xc2, 2Xc1)
(1Xc2, 2Xc1) b> (1Bc2, 2Bc1)
(1Bc2, 2Bc1) x> (1Xc2, 2Xc1)
(1Xc2, 2Xc1) b> (1Bc2, 2Bc2)
(1Bc2, 2Bc2) x> (1Xc2, 2Xc2)

Edges in intersection
(1Xc2, 2Xc2) c> (1Cc2, 2Cc2)
(1Xc2, 2Xc2) b> (1Bc2, 2Bc2)
Negation

- !(from A via B to C) = from A via !B to C
Negation

strategy:
! {A -> B
   B -> C}

class graph

start set

finish set

thick edges with incident nodes: traversal graph
Path expression problems

• Single-source path expression problem: Find for each vertex \( v \) an unambiguous path expression \( P(s,v) \) such that \( \sigma(P(s,v)) \) contains all paths from \( s \) to \( v \).

• Single sink path expression problem: Find for each vertex \( v \) an unambiguous path expression \( P(v,t) \) such that \( \sigma(P(v,t)) \) contains all paths from \( v \) to \( t \).
Path expression problems

• All-pairs path expression problem: Find for each vertex pair v,w an unambiguous path expression $P(v,w)$ such that $\sigma(P(v,w))$ contains all paths from v to w.
Negation

• Single-sink path expression problem for traversal graph. Not even needed?

• !(A->B->C->D) = A->!B->D | A->B->!C->D

• Each path from source to target must have a violation. Exponentially many paths?
Negation (base \{A->E\})

• \;!\{A->B1 A->B2 B1->C B2->C C->D1 C->D2 D1->E D2->E\} = 

• \{A->!B1 A->!B2 !B1-> E !B2->E\} | 

• \{A->B1 A->B2 B1->!C B2->!C !C->E\} | 

• \{A->B1 A->B2 B1->C B2->C C->!D1 C->!D2 !D1->E !D2->E\}
strategy:  
{A -> B  
B -> C} 

thick edges with incident nodes: traversal graph
Strategies and regular expressions

• Given a graph G with nodes s and t. We define a set of paths in G from s to t:
  – By a strategy
  – By a regular expressions. A any* B any* C = from A via B to C.

• Question: which model is better? More powerful?
Tarjan’s algorithm for negation

- Graph G, strategy S with source s and target t.
- Use Tarjan’s poly. time algorithm to construct the regular expression B of all paths from s to t.
- Turn S into a regular expression SR. Need to give details. Is it always possible?
- \( \neg S = B - SR \)
Implications of the Stockmeyer/Meyer paper regular-expression-like

- (Word problems requiring exponential time)
- $\text{INEQ}({0,1}, \{\text{union, concat}\}) \text{ NP-complete}$
- $\text{INEQ}({0}, \{\text{union, concat, star}\}) \text{ NP-comp.}$
- $\text{INEQ}({0,1}, \{\text{union, concat, compl}\}) \text{ name?}$
  Super exponential space
- $\text{Member}({0,1}, \{\text{union, concat, square, star, compl}\}) \text{ poly.}$
Implications of the Stockmeyer/Meyer paper

• Note: intersection is absent, use complement and union

• INEQ(\{0\}, \{union, concat, compl\}) poly
Strategy expressions and regular expressions

- \([A,B]\)                      \(A.\text{any}*.B\)
- through edges                  \(\text{any}*.\text{lnk}.*\text{any}^*\)
- bypassing edges                \(\text{not(}\text{any}*.\text{lnk}.*\text{any}^*)\)
- through vertices                \(\text{any}*.A.\text{any}^*\)
- bypassing vertices              \(\text{not(}\text{any}*.A.\text{any}^*)\)
- \(d_1\) join \(d_2\)            \([d_1].[d_2]\) (eliminate join point)
- \(d_1\) merge \(d_2\)          \([d_1]\cup[d_2]\)
- \(d_1\) intersect \(d_2\)      \([d_1]\cap[d_2]\)
- not \(d_1\)                     \(\text{not([d_1])}\)
- Only-through

??

intersection/negation
Strategy graph expressions versus strategy expressions

• With strategy graph expressions we start with strategy graphs and apply the set theoretic operators union, intersection, negation.

• With strategy expressions we start with single edge strategy graphs and apply operators.

• Are strategy graph expressions more expressive?
Example

Regular expression: ???

Regular expression:
(A any* B any* C)*
Goals

• Want to avoid spending an exponential amount of time in the size of the class graph (and strategy graph) to do a traversal on an object graph.

• Linear in size of object graph.
Goals

1. Fast type checking:
   1. To check whether PathSet[G](S) is empty is polynomial in G (and hopefully S).
   2. Quantified: To check whether PathSet[G](S) is empty for all G is exponential in S.

2. Fast traversal: The decision in the object graph whether to follow branch l after we have done prefix p is a constant time lookup in PathSet[G](S) (S considered a constant). PathSet[G](S) has a polynomial representation in G (and hopefully S).
Goals

1. Fast type checking:
   1. To check whether PathSet[G](S) is empty is polynomial in G (and hopefully S). ??
   2. Quantified: To check whether PathSet[G](S) is empty for all G is exponential in S.

2. Fast compilation: Compilation time should be polynomial time in the size of G and S.

3. Fast traversal: Given a startegy S and a class graph S and an object graph O, return the object graph O’ selected by S. This should be polynomial in the size of S, G and O.
Goals for PCDs

- As expressive and concise as possible to define path sets and subject to
  - Fast Type checking:
    - To check whether PathSet[G](S) is empty is polynomial in G (and hopefully S).
    - Quantified: To check whether PathSet[G](S) is empty for all G is exponential in S.
  - Fast Traversal: The decision in the execution tree whether to follow branch l after we have done prefix p is a constant time lookup in PathSet[G](S) (S considered a constant). PathSet[G](S) has a polynomial representation in G (and hopefully S).
Goals

• Conjecture: If we use strategy graphs with the operations (on path sets): union, intersection, complement, we have the desired properties.
Proof

• If we use strategy graphs with the operations (on path sets): union, intersection we have the desired properties.

• Union: easy. Intersection: assume only a constant number of intersections. Product of traversal graphs.
Add negation

- If we use strategy graphs with the operations (on path sets): union, intersection, negation we have the desired properties.

- Negation: Ravi or Tarjan+? Warning: the standard way of constructing the automaton for negation requires a deterministic automaton! (complement acceptance states).
Tarjan + turning strategies into regular expressions

• We also need to address goal 3: quick lookup where to go. If we have a complex regular expression and we need to know what the possible continuations are for prefix p, can we get it in constant time?

• If we have an NFA we can simulate it and see what the possible next states are. But for turning negation into an automaton, we get an exponential cost?
Regular expression complexity

• Determining whether a regular expression over \{0\} does not denote \(0^*\) is NP-complete.

• Strange: for a strategy: all paths from \(s\) to \(t = 0\).
Complexity of Compiling traversal strategies and pointcut designators

• Problem: Union-Join-Intersection-NonEmptyness
  – Input: Strategy graph S and class graph G.
  – Question: Is Pathset[G](S) nonempty?

• Union-Join-Intersection emptiness is NP-complete.
Proof

• We show the construction by an example. We start with a satisfiable boolean formula and translate it into a strategy and a class graph with source s and target t. The class graph is acyclic.

• The class graph is constructed in such a way that a path from s to t does not visit both x₁ and !x₁.
Boolean formula: 
\((x_1 + x_2)^*!x_1!*x_2\)

Strategy: 
\(\hspace{1em}([s,x_1][x_1,t] + [s,x_2][x_2,t])^*\) 
\(\hspace{1em}([s,!x_1][!x_1,t])^*\) 
\(\hspace{1em}([s,!x_2][!x_2,t])\)

Maximal tree of class graph:

Class graph:

!x_1 is a node in the strategy and class graph. All edges are optional.
Boolean formula:
\[(x_1 + x_2)\neg x_1\neg x_2\]

Strategy:
\[([s,x_1][x_1,t] + [s,x_2][x_2,t])\neg x_1\neg x_2\]
\[([s,!x_1][!x_1,t])\neg x_1\neg x_2\]
\[([s,!x_2][!x_2,t])\neg x_1\neg x_2\]

Maximal tree of class graph:

Class graph:

\[!x_1\text{ is a node in the strategy and class graph. All edges are optional.}\]
Boolean formula:
\((x_1 + x_2)*!x_1*!x_2\)

Strategy:
\([(s,x_1)[x_1,t] + [s,x_2][x_2,t])*\]
\(([s,!x_1][!x_1,t])*)\)
\(([s,!x_2][!x_2,t])\)

Maximal tree of class graph:

Class graph:
But the corresponding traversal problem is polynomial

- **Problem:** Union-Join-Intersection-Traversal
  - **Input:** Strategy graph S and class graph G and object tree O.
  - **Output:** The object tree O’ selected by S.
- **Union-Join-Intersection Traversal is polynomial.**
- **Proof:** current AP Library implementation.
Proof continued

• Instead of constructing the products, for each expression $S_1*S_2$ we construct the two traversal graphs and we simulate them at run-time.

• Notice that many of the traversals will terminate prematurely if the formula has few satisfying assignments.
Interesting connections

• It is surprising that Union-Join-Intersection Traversal is polynomial while Union-Join-Intersection Emptyness is NP-complete.

• The object tree represents the possible assignments. The traversal will try to find a path from s to t.

• The maximal tree is big but the size of it is part of the input.
The following problem is also NP-complete

- Problem: Union-Join-Intersection-Traversal-Success
  - Input: Strategy graph $S$ and class graph $G$.
  - Question: Is there an object graph $O$ so that $O'$ contains $t$.
- Union-Join-Intersection-Traversal-Success is NP-complete.
What is the complexity of:

- Negation-Union-Join-Intersection-Traversal-Success
- Negation-Union-Join-Intersection-Traversal
- Negation-Union-Join-Intersection-NonEmptyness
Justification

• Now we have justification for the current design of the AP Library.
  – If we would construct intersections of traversal graphs, the compiler would have to solve an NP-complete problem.
  – Are there still benefits to constructing the product?
Ravi’s construction of subtree

• Consider all leaves and all paths to them from the source.
• Check each path whether it satisfies strategy. The subtree is the union of all satisfying paths.
• This is polynomial.
NP completeness proofs

• Ravi’s proofs are in the wiki: node Satisfiability Hardness
Intersection and DAJ

• Intersection is central to DAJ.
• The problem: Consider the viewgraphs in:
  • The crucial one is shown on the next slide.
Which edges to follow to C2?

From o1 of class C1, follow edge e iff there is some object graph O and some o2, o3 s.t.
(1) e(o1,o2),
(2) O*(o2,o3), and
(3) class(o3) <= C2

The existential quantifier “there is some object graph” represents our lack of knowledge about the rest of the object graph.
The problem

• When we add intersection to the strategy language, the property formulated with an existential quantifier becomes NP-complete.
• So we can no longer ask the question: is there a subobject containing a target object by solving a graph reachability problem.
What is the new semantics?

- The current implementation in DAJ selects a subtree given a strategy $s_1*s_2$. The rule is to go down those edges where none of the strategies $s_1, s_2$ is violated with the information available so far. See next viewgraph for an example.

- But is this a subtree that is easily defined and easy to understand by programmers?
Traversal example showing subtree selected by DAJ

Boolean formula:
\((x_1 + x_2)^* \neg x_1^* \neg x_2\)

Strategy:
1. \([s,x_1][x_1,t] + [s,x_2][x_2,t]^*\)
2. \([s,\neg x_1][\neg x_1,t]^*\)
3. \([s,\neg x_2][\neg x_2,t]\)

Class graph:

Maximal tree of class graph:

The DAJ traversal will visit node \(\neg x_1\) besides the root \(s\). This seems arbitrary.
Possible reactions

• Limit the arguments of intersection: Allow only special kinds of strategies, for example: from A bypassing \{nodes,edges\} to B. The way intersection is used in DAJ is to “cut down the class graph”. We eliminate certain nodes and edges.

• Limit visitors: Can only have code on the Target class. This would be very restrictive.
Discussion

• “Limit the arguments to intersection” seems to be a good option. Intersection with elemental strategies (where the strategy is a subgraph of the class graph; called propagation graph in 1994) or strategies that expand to elemental strategies seems fine.

• We have done this in practice so far.
Discussion

• What is the impact of this on PCDs? Pengcheng and Jeff?
Traversing example showing subtree selected by DAJ

**Boolean formula:**
\((x_1 + x_2)!x_1!x_2\)

**Strategy:**
1. \([s,x_1][x_1,t] + [s,x_2][x_2,t]\) *
2. \([s,!x_1][!x_1,t]\) *
3. eliminated

**Class graph:**

```
  s  
   /  
  x1 /    
   |     
  !x1     
  / 
 t   
```

**Maximal tree of class graph:**

```
  :s  
  / 
 :x1 /   
  |  
 :!x1  
  / 
 :x2 /    
  |  
 :!x2  
  / 
 :t   
```

The DAJ traversal will visit the red path.

\(!x_1\) is a node in the strategy and class graph. All edges are optional.

This intersection is ok.

**Strategy:**
1. \([s,x_1][x_1,t] + [s,x_2][x_2,t]\) *
2. \([s, bypassing x_1,t]\) *
3. eliminated
New topic: Complexity of: given a strategy S, is there a class graph so that PathSet[G](S) is non-empty.

• Ravi proves this in the wiki: node Satisfiability Hardness

• Example follows.
Example: satisfiability of strategies

Boolean formula: \((x_1 + !x_2)!x_1\)

x_1 = false, x_2 = false

Strategy:
1. \(([s,x_1][x_1,t] + [s,\text{bypassing } x_2,t])^*\)
2. \(([s,\text{bypassing } x_1,t])^*\)

Class graph
Current activities (second half 2004)

Demeter Project

Incremental Software Development
(Growing software in small steps)

Organize inside increment

Law of Demeter:
Talk only to your friends.

Organize between increments

Law of Demeter for Concerns:
Talk only to your friends who share your concerns.

Aspect-Oriented Software Development (AOSD)

Improving AOSD

Better Selector Models

Contracts for Aspects

Design of Aspect Languages

Program chair AOSD 2004

Organizing chair AOSD 2003

Keynote speaker on AOSD

Dissemination of AOSD

Producing reliable Software with fewer vulnerabilities

Using AOSD for checking Security rules

Software security

Applying AOSD

Security as aspects

Program committee Memberships (AOSD Related)