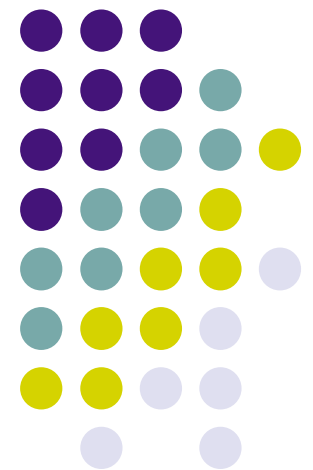
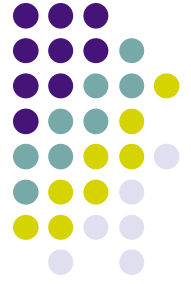


Selecting, Refining, and Evaluating Predicates for Program Analysis

Nii Doodoo, Lee Lin, Michel D. Ernst

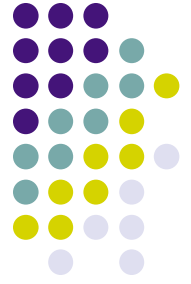


Presented by Kevin Do



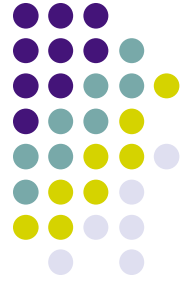
Outline

- Background
- Detecting implication
- Splitting condition & predicate
- Policies for selecting predicates
- Evaluation
- Conclusion



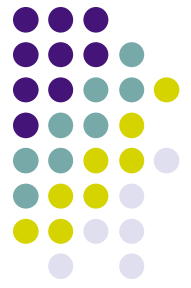
Introduction

- Program analysis is to determine the facts about the program
- The usefulness is depended on what properties it can report
- The challenge is increasing the grammar of program analysis without making the analysis more expensive and without degrading the quality of the output



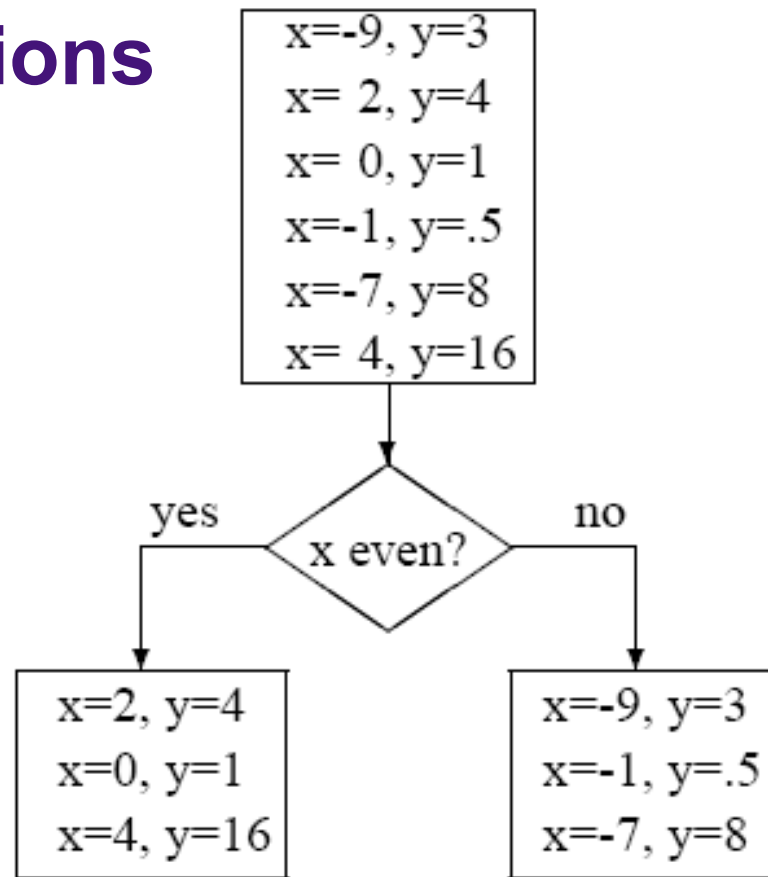
Introduction

- This paper proposed techniques for expanding the output grammar of a program analysis to include the implications (conditional properties)
- Implication form: $\mathbf{a} \Rightarrow \mathbf{b}$
 - Conditional program property is the one whose consequent is true when the predicate is true.
 - It is infeasible to check for every \mathbf{a} and \mathbf{b}
- **How to improve it ?**
 - Restrict what properties are used for \mathbf{a}
 - Permit \mathbf{b} to range over all properties that report by the analysis
- Idea: use splitting conditions (partition data) and then combine separate analysis to create implications



Detecting implications

1. Split the data into parts



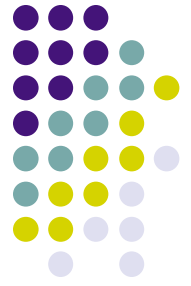
2. Compute properties over each subset of data

x even
 $x \geq 0$
 $y = 2^x$
 $y > 0$

x odd
 $x < 0$
 $y > 0$

3. Compare results, produce implications

$x \text{ even} \Leftrightarrow x \geq 0$
 $x \text{ even} \Rightarrow y = 2^x$
 $x \geq 0 \Rightarrow y = 2^x$



Algorithm for creation of implication

// S_1 and S_2 are sets of properties resulting from
// analysis of partitions of the data.

procedure CREATE-IMPLICATIONS(S_1, S_2)

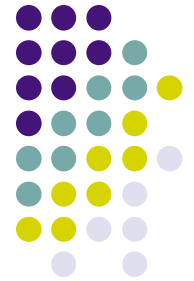
for all $p_1 \in S_1$ **do**

if $\exists p_2 \in S_2$ such that $p_1 \Rightarrow \neg p_2$ and $p_2 \Rightarrow \neg p_1$ **then**

// p_1 and p_2 are mutually exclusive

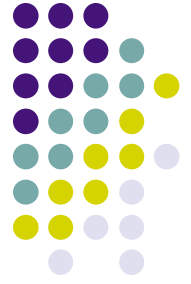
for all $p' \in (S_1 - S_2 - \{p_1\})$ **do**

output “ $p_1 \Rightarrow p'$ ”



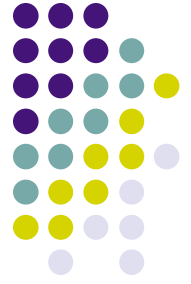
Creation of implication

Properties		Implications		Simplified
S_1	S_2	$a \Rightarrow b$	$\neg a \Rightarrow \neg b$	$a \Leftrightarrow b$
a	$\neg a$	$a \Rightarrow d$	$\neg a \Rightarrow f$	$a \Rightarrow d$
b	$\neg b$	$a \Rightarrow e$	$\neg b \Rightarrow \neg a$	$a \Rightarrow e$
c	c	$b \Rightarrow a$	$\neg b \Rightarrow f$	$\neg a \Rightarrow f$
d	f	$b \Rightarrow d$		
e		$b \Rightarrow e$		



Splitting conditions & predicates

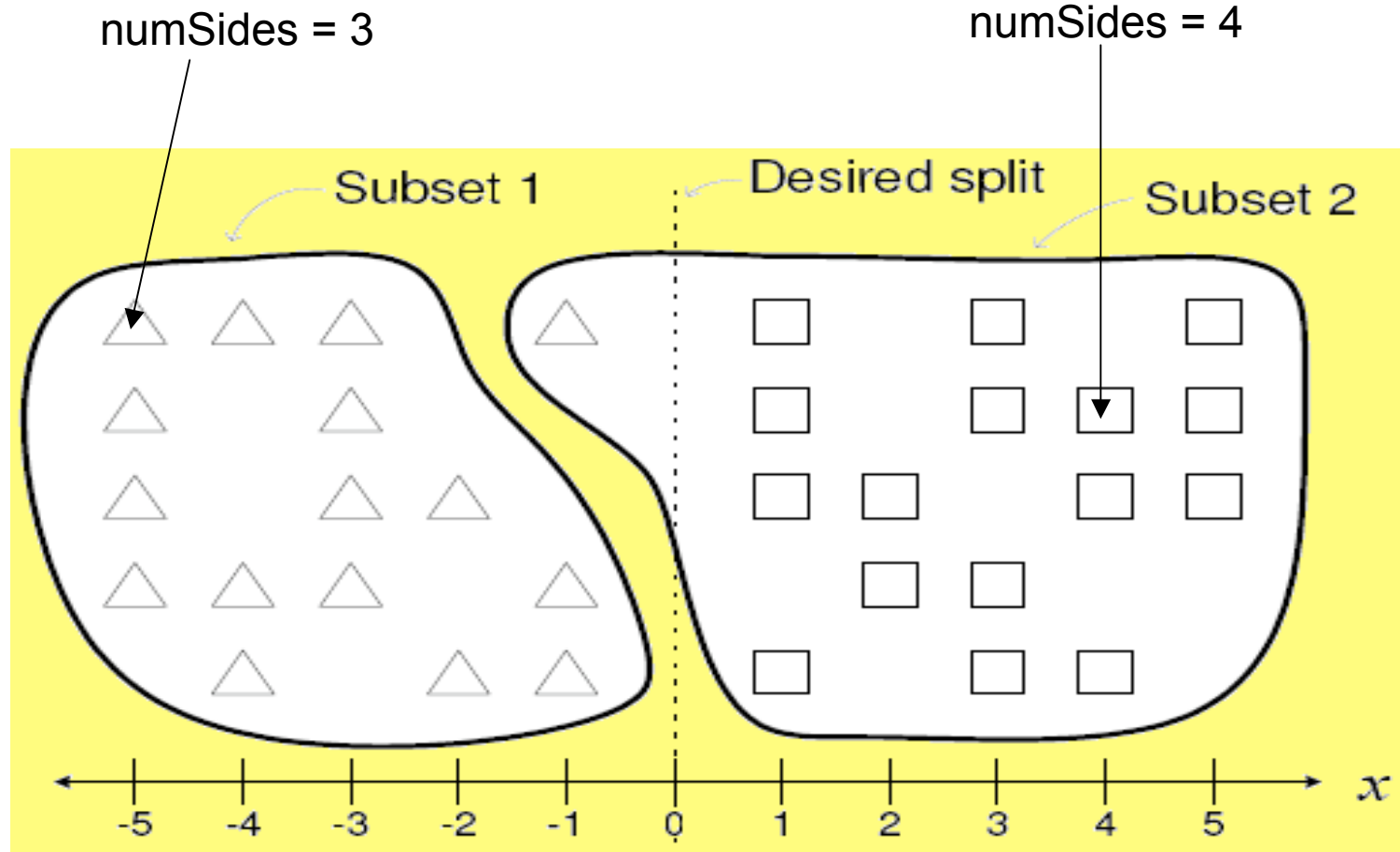
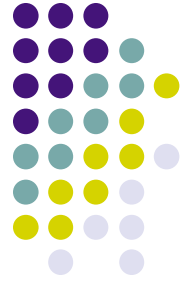
- Split condition appears as left-hand-side, b/c it is guaranteed to be true of one subset of the data (and likewise for its negation)
- Split condition might not be reported in a subset of the data:
 - Split condition may be inexpressible in the analysis tool's output grammar
 - A stronger condition may be detected; the weaker implied property need not to be reported
 - Split condition may not be statically justified



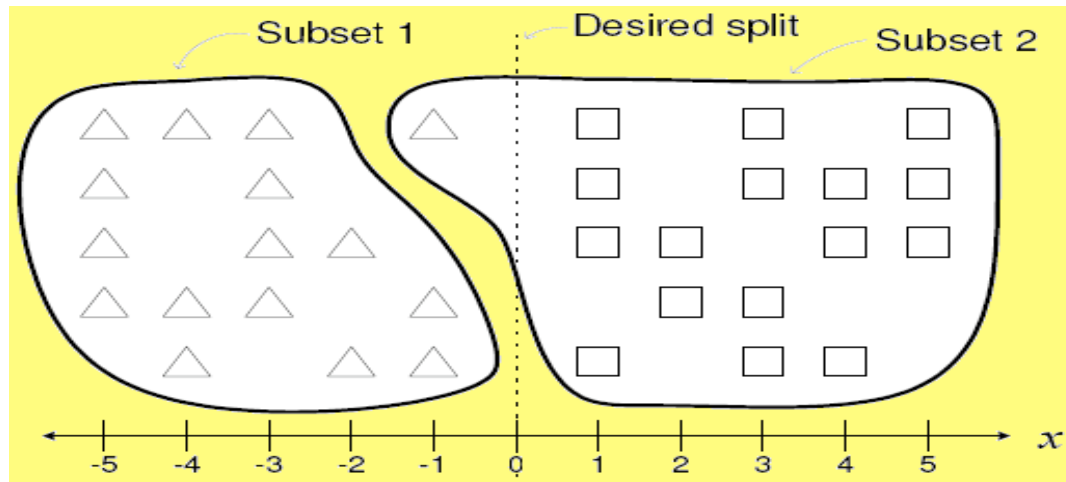
Refining splitting conditions

- Splitting may be good but not perfect partition of the data
- Need two-pass process that performs the analysis twice
 1. First pass to produce a refined set of splitting conditions
 2. Second pass is to output implication

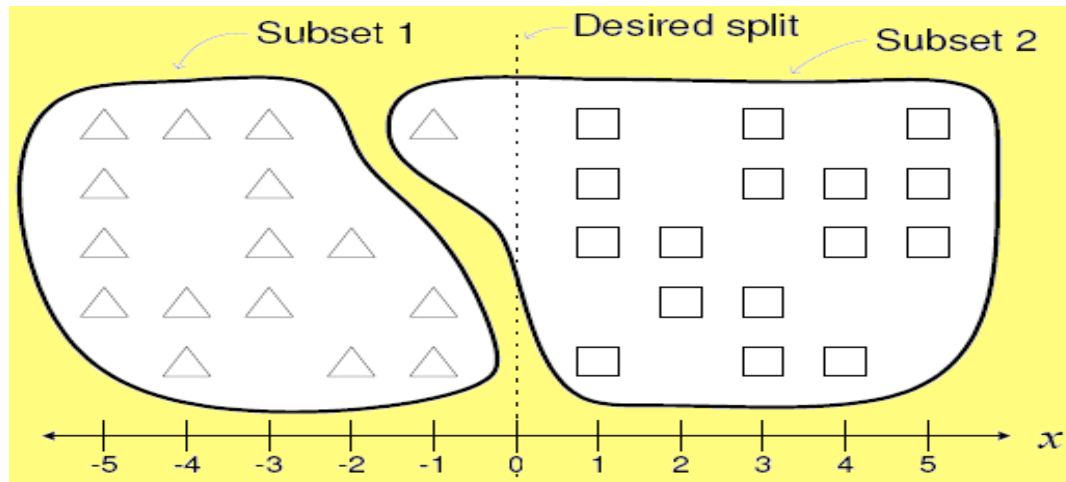
Example



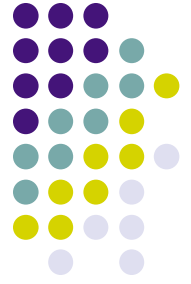
Initial split into two subsets nearly, but not exact



- 1st pass: using the subsets as the splitting condition
 - subset = 1 \Rightarrow numSides = 3
 - subset = 1 \Rightarrow $x < 0$
- Refined splitting condition:
 - numSides = 3 and $x < 0$



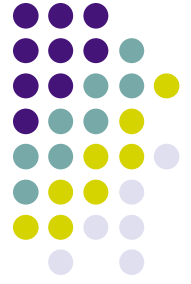
- **2nd pass**
 - $(x > 0) \Rightarrow (\text{numSides} = 4)$
 - $(x < 0) \Rightarrow (\text{numSides} = 3)$



How to select predicates

Policies:

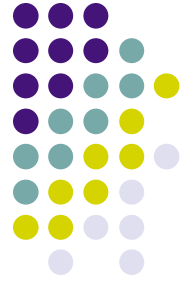
- Procedure return analysis
- Static analysis for code conditionals
- Clustering
- Random selection



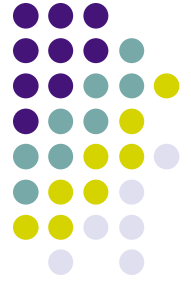
Procedure return analysis

- This policy based on 2 dynamic checks:
 - First check splits data based on the return site
 - Second check splits data based on boolean return value

Static analysis for code conditionals

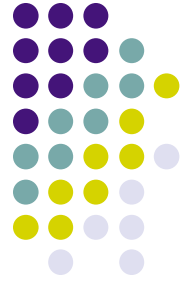


- Select each boolean condition used in the program (as the test of a *if*, *while*, or *for* statements, or as the body of a pure boolean member of function) as a splitting condition



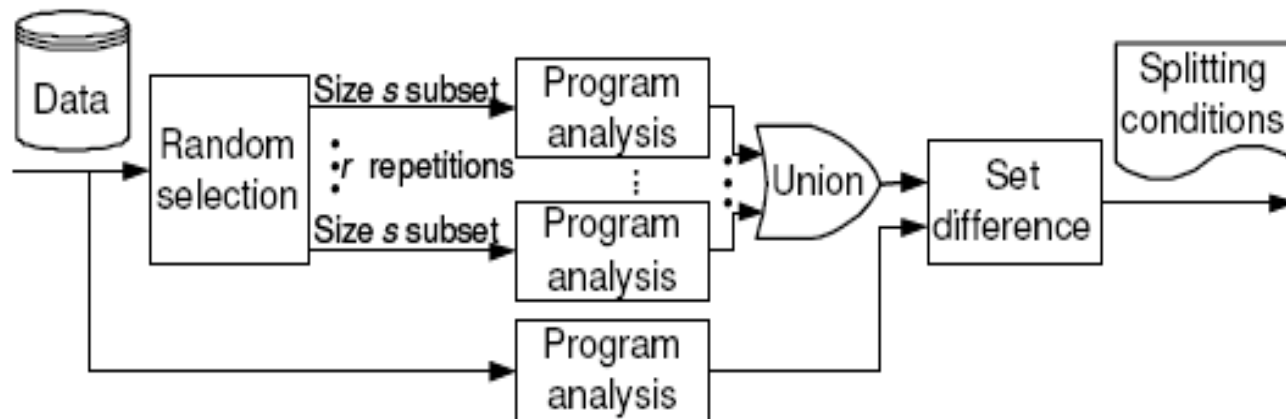
Clustering

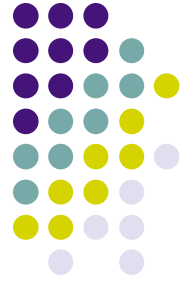
- Partition data points into clusters that are internal (members in same group) and external (members in different group)
- This policy uses a two-pass algorithm to refine its inherently approximate results



Random selection

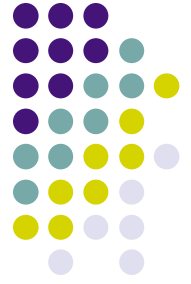
- Select r different subsets of the data (size s) then apply program analysis over each subset
- Use any property detected in one of the subset as a splitting condition





Evaluation

- *Daikon: dynamic invariant detection*
- *ESC/Java: static checker to verify lack of runtime errors*
- **Goal set** is the properties that discriminate between faulty and non-faulty runs
- **Reported set** is the consequents of conjectured implications
- **Matching set** is the intersection between goal and reported set



Evaluation

Goals of this experimental evaluation:

- **Static Checking:** measure the accuracy of program analysis results for a program verification task.
- 2. **Error detection:** measure how well the implication indicated faulty behavior

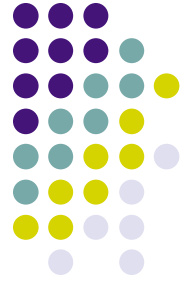
- ***Precision and Recall***
 - *Precision: measure of correctness*
 - Precision = matching / reported
 - *Recall: measure of completeness*
 - Recall = matching / goal



Static Checking

Program	Program size		No implications		Return		Static		Cluster		Random	
	LOC	NCNB	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
FixedSizeSet	76	28	1.00	0.86	1.00	0.86	1.00	0.86	1.00	0.86	1.00	0.86
DisjSets	75	29	0.82	1.00	1.00	0.97	1.00	1.00	1.00	0.94	0.80	0.98
StackAr	114	50	1.00	0.90	1.00	1.00	0.95	1.00	0.78	1.00	0.95	1.00
QueueAr	116	56	0.92	0.71	0.98	0.78	0.89	0.84	0.62	0.89	0.77	0.91
Graph	180	99	0.80	1.00	0.80	1.00	0.80	1.00	0.80	1.00	0.80	1.00
GeoSegment	269	116	1.00	1.00	1.00	1.00	1.00	1.00	0.94	1.00	0.73	1.00
RatNum	276	139	0.93	1.00	0.91	1.00	1.00	1.00	0.72	1.00	0.50	1.00
StreetNumberSet	303	201	0.82	0.95	0.77	0.95	0.77	0.96	0.77	0.96	0.83	0.89
Vector	536	202	0.96	0.95	0.99	0.95	0.76	0.98	0.71	0.97	0.81	0.97
RatPoly	853	498	0.81	0.97	0.67	0.95	0.71	0.96	0.68	0.96	0.79	0.95
Total	4886	2451	0.91	0.93	0.91	0.95	0.89	0.96	0.80	0.96	0.80	0.96
Missing			0.09	0.07	0.09	0.05	0.11	0.04	0.20	0.04	0.20	0.04

Figure 9. Invariants detected by Daikon and verified by ESC/Java, using four policies for selecting splitting conditions (or using none). “LOC” is the total lines of code. “NCNB” is the non-comment, non-blank lines of code. “Prec” is the precision of the reported invariants, the ratio of verifiable to verifiable plus unverifiable invariants. “Recall” is the recall of the reported invariants, the ratio of verifiable to verifiable plus missing. “Missing” indicates the overall missing precision or recall. The most important measure is the missing recall (in bold): it is the most accurate measure of human effort, since it indicates how much humans must add to the reported set. By comparison, removing elements from the set — measured by the complement of precision — is an easy task.



Error detection

- To help locating the program error
- Errors induce different program behaviors (program behaves differently on erroneous runs than on correct runs)

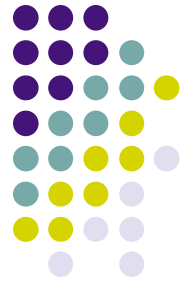


Result of error detection

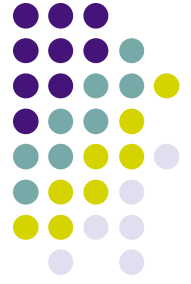
Program	Source	Program size			Return		Static		Cluster		Random	
		Ver.	LOC	NCNB	Prec.	Recall	Prec.	Recall	Prec.	Recall	Prec.	Recall
NFL	TC	10	23	21	0.00	0.00	0.03	0.08	0.03	0.08	0.09	0.37
Contest	TC	10	21	17	0.00	0.00	0.19	0.40	0.11	0.23	0.15	0.21
Azot	TC	6	18	17	0.00	0.00	0.00	0.00	0.13	0.46	0.12	0.15
RatPoly	MIT	32	853	498	0.03	0.00	0.03	0.01	0.07	0.03	0.14	0.09
CompostiteRoute	MIT	67	883	319	0.22	0.09	0.22	0.09	0.21	0.47	0.21	0.45
print_tokens	S	7	703	452	0.00	0.00	0.03	0.13	0.04	0.22	0.04	0.34
print_tokens2	S	10	549	379	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
replace	S	32	506	456	0.24	0.31	0.14	0.28	0.10	0.42	0.15	0.37
schedule2	S	9	369	280	0.00	0.00	0.20	0.35	0.20	0.35	0.24	0.41
tcas	S	41	178	136	0.17	0.24	0.19	0.31	0.12	0.23	0.13	0.40
tot_info	S	23	556	334	0.00	0.00	0.12	0.27	0.09	0.40	0.03	0.09
Total		247	137015	75115	0.06	0.06	0.10	0.18	0.10	0.26	0.12	0.26

Figure 12. Detection of conditional behavior induced by program errors, compared for four splitting policies. (When no splitting is in effect, the precision and recall are always zero.) “Ver” is the number of versions of the program. “LOC” is the average total lines of code in each version. “NCNB” is the average non-comment, non-blank lines of code. “Prec” is the precision of the reported invariants, the ratio of matching to reported. “Recall” is the recall of the reported invariants, the ratio of matching to goal. In this experiment, precision (in bold) is the most important measure: it indicates how many of the reported implications indicate erroneous behavior. In cases where precision is 0.00, the experiment did report some implications, but none of them contained consequents in the goal set.

Evaluation – comparing policies



1. The procedure return analysis highly depend on particular implementation chosen by the programmer.
2. The code conditional policy is good for ESC/Java b/c it verifies the program source code, and it needs to prove conditions having to do with code paths through conditional.
3. Clustering policy: structure in a vector space composed of values from the traces being analyzed.
4. Random selection policy does not work well if there is a relatively equal split between behaviors, and cannot take advantage of structure when it is present (from source code)



Conclusion

- 4 policies that can be used in conjunction with the implication technique. No policy dominates any other, the last two perform the best on average.
- Improve performance on verification task (reduce # of missing properties)
- Detect difference in behavior between faulty and non-faulty program runs