The Concept of Dynamic Analysis

Thomas Ball

Bell Laboratories
Lucent Technologies

Dynamic Analysis vs Static Analysis

- Static Analysis
  - Examines a program's text to derive properties hold for all executions
- Dynamic Analysis
  - Examines the running program to derive properties hold for one or more executions
  - Detects violations of properties
  - Provides useful information

Dynamic Analysis vs Static Analysis

- Complementary Techniques
  - Completeness
    - Dynamic analysis generate "dynamic program invariants" for the observed set of executions
    - Static analysis helps to determine "dynamic program invariants" true or not for all program executions
    - Cause of disagree
      - Not sufficient executions for dynamic analysis
      - Examining infeasible paths in static analysis

Dynamic Analysis vs Static Analysis

- Complementary Techniques (cont)
  - Scope
    - Dynamic analysis potentially discovers “dependencies at a distance”
    - Static analysis has difficulties (restricted in scope)to do so
  - Precision
    - Dynamic analysis examines the concrete domain of program execution
    - Static analysis abstracts over this domain to ensure termination of the analysis, losing information from start.

Usefulness of Dynamic Analysis

- Precision of information
  - Particular execution to collect precise information to address particular problems
- Dependence on program inputs
  - Relate program input and output to program behavior

Two Dynamic Analysis

- This paper proposes two dynamic analysis
  - Frequency Spectrum Analysis (FSA)
  - Coverage Concept Analysis (CCA)
  - Both are based on program profile

Frequency Spectrum Analysis (FSA)

- Goal
  - Analyzing the frequencies of the program entities in a single execution to help programmers to
    - decompose a program;
    - identify related computations;
    - find computations related to specific input and output characteristic of the program

Frequencies & Program Behavior

- Low Frequencies vs High Frequencies
  - Execution frequencies of program entities implies their place in the hierarchy of program abstraction
  - Example: sorting module
    - Interface procedures execute many fewer times than private procedures that invoke one anther to perform sorting operation

Frequency & Program Behavior (cont)

- Related Frequencies and Frequency Clusters
  - What are Frequency clusters
  - Put entities together through common frequency and implies their dynamic relationship
- Specific Frequency
  - Frequencies related to input/output implies parts of program responsible for input/output
  - Example
    - An enumeration of record as output might imply the frequency of a program entity in size of the enumeration

Case Study

- Apply FSA to profile of an example (obfuscated C program)
- Restructuring a C program based on the result of FSA

Example Input & Output

- Example: take no input, print out a poem “The Twelve Days of Christmas”

```
On the first day of Christmas my true love gave to me
a partridge in a pear tree.

On the second day of Christmas my true love gave to me
two turtle doves
and a partridge in a pear tree.

...

On the twelfth day of Christmas my true love gave to me
twelve drummers drumming, eleven pipers piping, ten lords a-leaping,
nine ladies dancing, eight maids a-milking, seven swans a-swimming,
six geese a-laying, five gold rings;
four calling birds, three french hens, two turtle doves
and a partridge in a pear tree.
```

Fig. 5. Partial output of the obfuscated C program.

Example Source Code

- Obfuscated C Program

```
#include <stdio.h>
main(t,_,a)char*a;{
return!0<t?t<3?main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a)):
1,t<_?main(t+1,_,a):3,main(-94,-27+t,a)&&t==2?_<13?
main(2,_+1,"%s %d %d\n"):9:16:t<0?t<-72?main(_,t,
"@n'+,#'/*{}w+/w#cdnr/+,{}r/*de}+,/*{*+,/w{%+,/w#q#n+,/#{l+,/n{n+,/+#n+,/#\
;#q#n+,/+k#;*+,/'r :'d*'3,}{w+K w'K:'+}e#';dq#'l \
q#'+d'K#!/+k#;q#'r}eKK#}w'r}eKK{nl]'/#;#q#n'){)#}w'){){nl]'/+#n';d}rw' i;# \
){nl]!/n{n#'; r{#w'r nc{nl]'/#{l,+'K {rw' iK{;[{nl]'/w#q#n'wk nw' \
iwk{KK{nl]!/w{%'l##w#' i; :{nl]'/*{q#'ld;r'}{nlwb!/*de}'c \
;;{nl'-{}rw]'/+,}##'*}#nc,',#nw]'/+kd'+e}+;#'rdq#w! nr'/ ') }+}{rl#'{n' ')# \
}'+}##(!!/")
:t<-50?_==*a?putchar(31[a]):main(-65,_,a+1):main((*a=='/')+t,_,a+1)
:0<t?main(2,2,"%s"):*a=='/'||main(0,main(-61,*a,
"!ek;dc i@bK'(q)-[w]*%n+r3#l,{}:\nuwloca-O;m .vpbks,fxntdCeghiry"),a+1);}
```

Fig. 1. An obfuscated C program to print the poem "The Twelve Days of Christmas". The partial output of the program is shown in Figure 5.

Program Behavior

- Path Profile

Path ID	Frequency	Path ID	Frequency
main:0	1	main:2	114
main:19	1	main:3	114
main:22	1	main:1	2358
main:23	10	main:7	2358
main:9	11	main:4	24931
main:13	55	main:5	39652

Table 1. A path profile of the (readable) obfuscated C program's execution.

Other Version of Source Code

- Readable Version

```
#include <stdio.h>
main(t,_,a) char *a;
{
       if ((!0) < t) {
[1]      if (t < 3)  main(-79,-13,a+main(-87,1-_,main(-86,0,a+1)+a));
[2]      if (t < _ ) main(t+1,_,a);
[3]      main(-94,-27+t,a);
[4]      if (t==2 && _ < 13 ) main(2,_+1,"");
       } else if (t < 0) {
[5]      if (t < -72)  main(_,t,LARGE_STRING);
         else if (t < -50 ) {
[6]        if (_ == *a) putchar(31[a]);
[7]        else          main(-65,_,a+1);
[8]      } else main((*a=='/')+t,_,a+1);
[9]    } else if (0 < t) main (2,2,"%s");
[10]   else if (*a!='/') main(0,main(-61,*a,SMALL_STRING),a+1);
}
```

Fig. 2. A (more) readable version of the obfuscated C program, after reformatting, performing local syntactic substitutions to turn expressions into statements and eliminating dead code. There are 10 lines containing calls, each uniquely numbered in brackets.

Profile Information

- Summary information

Path ID	Frequency	Condition	Call Lines
main:0	1	`t == 1`	[9]
main:19	1	`t==2 && t >= _`	[1,3,4]
main:22	1	`t==2 && t < _ && _ >= 13`	[1,2,3]
main:23	10	`t==2 && t < _ && _ < 13`	[1,2,3,4]
main:9	11	`t >= 3 && t >= _`	[3]
main:13	55	`t >= 3 && t < _`	[2,3]
main:2	114	`t == 0 && *a == '/'`	*no call lines*
main:3	114	`t < -72`	[5]
main:1	2358	`t == 0 && *a != '/'`	[10]
main:7	2358	`t > -72 && t < -50 && _ == *a`	[6]
main:4	24931	`t < 0 && t >= -50`	[8]
main:5	39652	`t > -72 && t < -50 && _ != *a`	[7]

Table 2. Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.

Output Structure

- Poem’s nature structure
  - 12 verses for 12 days
  - 26 unique strings
    - 3 common strings, “on the”, “day of Christmas…”, “and a partridge”
    - 12 strings for ordinals (first, second, third, …, twelfth)
    - 11 strings for second through twelve gifts
  - 66 occurrences of presents other than “partridge in a pear tree”
    - 0+1+2+…+11 = 66
  - 114 strings printed
    - 12 occurrences of 3 common strings (12*3=36)
    - 12 ordinals
    - 66 non-partridge gifts
  - 2358 characters printed

Output & Program Behavior

- Correlation between the poem’s nature and program profile data
  - Execution count (frequency) implies responsible part of the program
  - Example
    - Main:7 path execution 2358 implies this path is corresponding to 2358 characters printing
  - Idea to reconstruct the C program

FSA on Program Profile

- Closer Examination on The Code and Table 2
  - 6 path cluster groups

Path ID	Frequency	Condition	Call Lines
main:0	1	t == 1	[9]
main:19	1	t==2 && t >= _	[1,3,4]
main:22	1	t==2 && t < _ && _ >= 13	[1,2,3]
main:23	10	t==2 && t < _ && _ < 13	[1,2,3,4]
main:9	11	t >= 3 && t >= _	[3]
main:13	55	t >= 3 && t < _	[2,3]
main:2	114	t == 0 && *a == '/'	*no call lines*
main:3	114	t < -72	[5]
main:1	2358	t == 0 && *a != '/'	[10]
main:7	2358	t > -72 && t < -50 && _ == *a	[6]
main:4	24931	t < 0 && t >= -50	[8]
main:5	39652	t > -72 && t < -50 && _ != *a	[7]

Table 2. Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.

- Path main:0 (initialization, execute once)
- Paths main:19, main:22, main:23 (1+1+10 = 12 verses) make up the outer loop;
- Paths main:9, main:13 (11+55 = 66 non-partridge-gifts within a verse) make up the inner loop;
- Paths main:2, main:3(114, 114) print out the 114 strings;
- Paths main:1, main:7 (2358, 2358) print out 2358 characters.

Restructure

- Restructure Program based on the FSA
  - main (path main:0)
    - initialization
  - outer_loop (paths main:19, main:22, main:23)
    - 12 verses
  - inner_loop (paths main:9, main:13)
    - 66 non-partridge-gifts within a verse
  - print_string (paths main:2, main:3)
    - 114 strings
  - output_chars (paths main:1, main:7)
    - print out 2358 characters
  - translat_and_put_char (path main:5)
  - skip_n_strings (path main:4)
- Source code not shown

Result

- Restructured program profile (table 3) and old program profile (table 2) in next slide
- The restructured program has the exact output with the original program

Comparison of Profiles

Path ID	Frequency	Path ID	Frequency
main:0	1	skip_n_strings:0	114
outer_loop:0	1	skip_n_strings:2	1898
outer_loop:1	11	output_chars:0	2358
inner_loop:0	12	translate_and_put_char:2	2358
inner_loop:1	66	skip_n_strings:1	23033
output_chars:1	114	translate_and_put_char:0	39652
print_string:0	114		

Table 3. The path profile of the restructured program.

Path ID	Frequency	Condition	Call Lines
main:0	1	`t == 1`	[9]
main:19	1	`t==2 && t >= _`	[1,3,4]
main:22	1	`t==2 && t < _ && _ >= 13`	[1,2,3]
main:23	10	`t==2 && t < _ && _ < 13`	[1,2,3,4]
main:9	11	`t >= 3 && t >= _`	[3]
main:13	55	`t >= 3 && t < _`	[2,3]
main:2	114	`t == 0 && *a == '/'`	*no call lines*
main:3	114	`t < -72`	[5]
main:1	2358	`t == 0 && *a != '/'`	[10]
main:7	2358	`t > -72 && t < -50 && _ == *a`	[6]
main:4	24931	`t < 0 && t >= -50`	[8]
main:5	39652	`t > -72 && t < -50 && _ != *a`	[7]

Table 2. Summary of the twelve executed paths in the readable obfuscated C program of Figure 2.

Summary

- FSA features
  - Partition the program by levels of abstract based on frequency;
  - Identify related computation based on frequency cluster;
  - Find computation related to the program's behavior based on specific frequency.

Unanswered Questions

- Shortcoming of the example
  - Direct relationship between the program's output and program's behavior
  - Size of the profile
  - No input

Coverage Concept Analysis (CCA)

- Concept analysis
  - Techniques to identify groups of objects that have common attributes
  - Input to concept analysis (binary relation)
    - Example (Test coverage table)

	Procedures					
Test	**add**	**lRotate**	**rem**	**Min**	**Succ**	**DelFix**
t1	X		X			X
t2	X	X	X			X
t3	X	X	X	X	X	
t4	X	X	X	X	X	X
t5	X	X	X	X	X	X

Definition

- Pair (T, E), where T is a set of tests and E is a set of program entities, is a *concept* if every test in T cover all entities in E, and no test outside T covers all entities in E.
- *Concept* determine maximal sets of tests covering identical entities(and maximal sets of entities covered by identical tests)

Example

	Procedures					
Test	add	lRotate	rem	Min	Succ	DelFix
t1	X		X			X
t2	X	X	X			X
t3	X	X	X	X	X	
t4	X	X	X	X	X	X
t5	X	X	X	X	X	X

Concept	Tests	Procedures
c1	t4, t5	add, lRotate, rem, Min, Succ, DelFix
c2	t3, t4, t5	add, lRotate, rem, Min, Succ
c3	t2, t4, t5	add, lRotate, rem, DelFix
c4	t2, t3, t4, t5	add, lRotate, rem
c5	t1, t2, t4, t5	add, rem, DelFix
c6	t1, t2, t3, t4, t5	add, rem

Partial Order

- Partial order

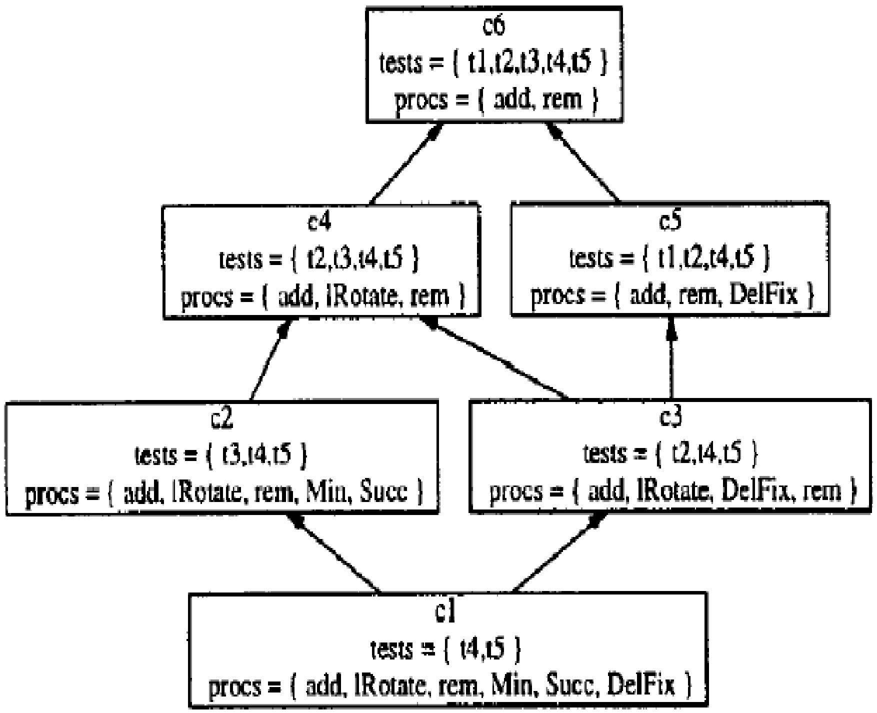
$$(T_1, E_1) \sqsubseteq (T_2, E_2) \iff T_1 \subseteq T_2 \iff E_2 \subseteq E_1$$

- Concept lattice

Concept Lattice

Example

Concept	Tests	Procedures
c1	t4, t5	add, lRotate, rem, Min, Succ, DelFix
c2	t3, t4, t5	add, lRotate, rem, Min, Succ
c3	t2, t4, t5	add, lRotate, rem, DelFix
c4	t2, t3, t4, t5	add, lRotate, rem
c5	t1, t2, t4, t5	add, rem, DelFix
c6	t1, t2, t3, t4, t5	add, rem

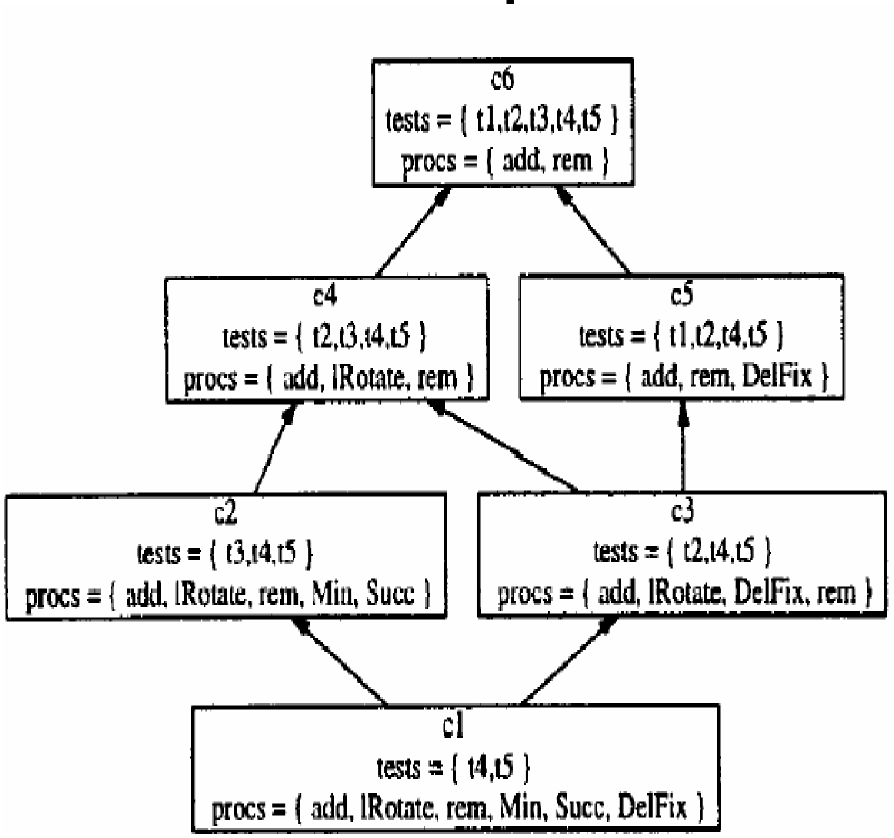


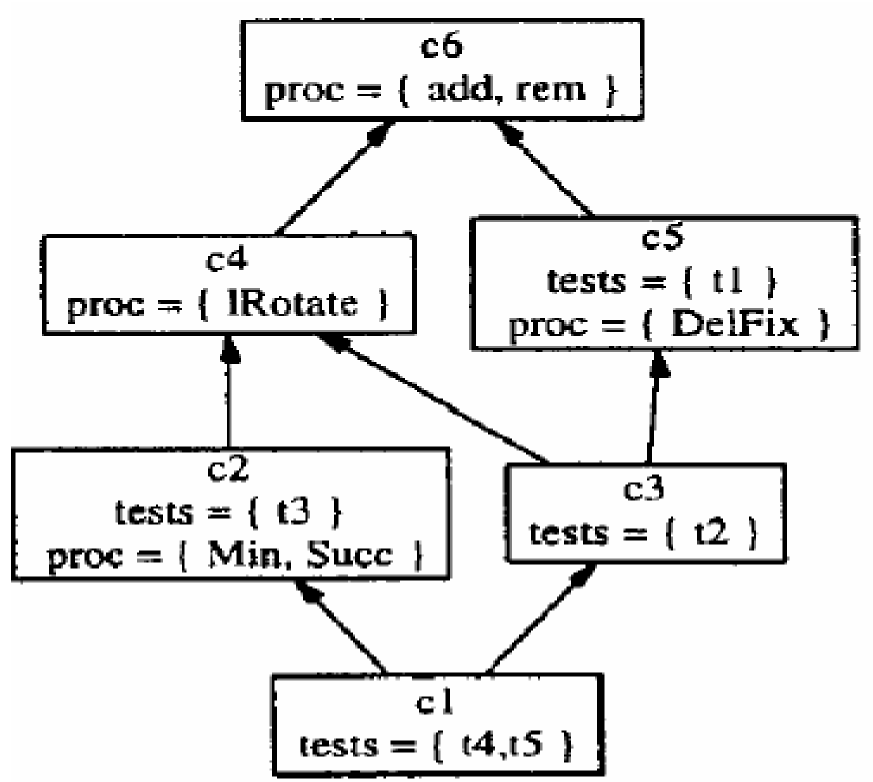
Concept Lattice Properties

- If test *t* is in a concept *c*, the t is in any concept greater than *c* . If entity *e* is in a concept *c*, then e is in any concept less than *c*.
- For every test *t*, there is a unique least concept *c* in which it appears, denoted by *lcont(t)*. For every entity *e*, there is a unique greatest concept *c* in which it appears, denoted by *gcont(e)*.

Least Concept and Greatest Concept

- Example





CCA Contribution

- Analog to Static Control Flow Relationships
  - Domination, Postdomination and Region
- Identifies “Dynamic Control Flow Invariant”

Domination, Postdomination & Control Flow Implication

- Definition of domination and postdomination
  - Entity *e* is said to dominate entity *f* if every path from program entry to *f* includes *e*.
  - Entity *f* is said to postdominate entity *e* if every path from *e* to program exit includes entity *f*.

Domination, Postdomination & Control Flow Implication(cont)

- Control Flow Implication
  - If entity *f* is in a concept greater than or equal to *gcon(e)*, then the execution of *e* dynamically implies the execution of *f*, which means in this test *f* dynamically dominate *e*.

Regions

- If entity *e* dominate *f* and *f* postdominate *e*, *e* and *f* are in the same region.
- By the concept lattice, if *gcon(e)* = *gcon(f)* then e and f are in the same dynamic region.

Dynamic & Static Information

- Dynamic information may not imply static information.
- Static information always implies dynamic information.

Conclusion

- FSA and CCA can aid in the tasks of program comprehension, program restructuring and new test development.