Locating Faults Through Automated Predicate Switching

Authored by **Xiangyu Zhang, Neelam Gupta, Rajiv Gupta**

The University of Arizona

ICSE 2006

Presented by **Jing Pu**

Authors

Xiangyu Zhang

University Scholar
Associate Professor
Department of Computer Science
Purdue University
Phone: 765-496-9415
E-mail: xyzhang at cs.purdue.edu

Neelam Gupta

The University of Arizona

Rajiv Gupta

Professor
Dept. of Computer Science & Engineering
University of California Riverside
408 Winston Chung Hall
Riverside, CA 92521, USA
Voice: (951) 827-2558
Fax: (951) 827-4643
E-mail: **gupta@cs.ucr.edu**

Outline

- Motivation
- Approach
- Key Techniques
- Limitations
- Evaluation
- Conclusion

Motivation

- Finding Bugs
  - Program output deviates from the expected output
  - Program crash

- Techniques

  Automated debugging techniques – **potential state changes searching is expensive**

- Idea
  - Switch outcome of some predicate at runtime – **correct erroneous output / eliminate crash** – Important information
  - Reduce the state search space - **runtime predicate switching**

Motivation Example


```
1.  read(a,b);
2.  c = f(a,b);
3.  if c < 5
4.  then x=a+b
5.  else x=a-b
6.  endif
7.  d = g(a,b);
8.  if d < 5
9.  then y=a*b
10. else y=a/b
11. endif
12. output(x+y)
```

Why predicate Switching is a better approach?

- How to compute an output – Two parts: Data Part, Select Part
- ***Data Part***: Set of executed instructions which compute data values that are involved in computing output value. <1,4,5,9,10>
- ***Select Part:*** Set of predicates and statements that compute values used by the predicates. <1,2,7,3,8>
- ***Location of fault code:***
  - DP: dynamic data slices are relatively ***small***
  - SP: union the full dynamic slices, consider all possible state changes — ***large***
- ***Switching predicate*** – Overcome this problem

Approach

- **Step 1:** Examine failing run, determine output deviation between incorrect and correct output, Locate execution instance I_e that produces the first erroneous output value.
- **Step 2:** Rerun program, generate Predicate Trace, perform predicate instance ordering – LEFS/PRIOR
- **Step 3:** Search for a critical predicate, for each predicate instance(*P*), rerun the program, use instrumented program to switch *P*'s output (only one predicate switch at a time), terminate search when program run succeeds
- **Step 4:** Based on critical predicate information, use Bidirectional Chop to locate the set of potentially faulty statements.

Predicate Instance Ordering

- Last Executed First Switched Ordering (LEFS)

  Based on the observation: execution of faulty code is often not far away from the fail point

- Prioritization-based Ordering (PRIOR)
  - Partition predicates into **high** and **low** priority subsets using failure- Inducing chops algorithm
  - Arrange the predicate instances in the high priority subset in the order of increasing dependence distance rom the erroneous output

- **Reference: N Gupta, H He, X Zhang, R Gupta, "Locating Faulty Code Using Failure-Inducing Chops", ASE 2005**

```
1  main(int argc, char *argv[ ])
2  {
3      int red, green, blue, yellow;
4      int sweet,sour,salty,bitter;
5      int i;
6
7      red = atoi (argv[1]);
8      blue = atoi (argv[2]);
9      green = atoi (argv[3]);
10     yellow = atoi (argv[4]);
11
12     red = 2*red;
13     sweet = red*green;
14     sour = 0;
15     i = 0;
16     while ( i < red) {
17           sour = sour + green;
18           i = i + 1;
19      }
20      salty = blue + yellow;
21      yellow = sour + 1;
22      bitter = yellow + green;
23
24     printf ("%d %d %d %d\n", bitter,sweet,
25                 sour,salty);
26     return 0;
27 }
```

Error: red = 5*red (

Initial Inputs: *input1* : = [1,5,8,2] - ×
input2 : = [0,0,0,0] - √

1-minimal failure-inducing inputs: *input3* : = [1,0,8,2] - ×
input4 : = [0,0,8,2] - √

argv[1] is different

Incorrect outputs at line 24: *bitter, sweet, sour*

$FwdSlice(input3, argv[1]) = \{7, 12, 13, 16, 17, 18, 21, 22, 24\}$

Forward slice on argv[1]

$BwdSlice(input3, bitter@24) = \{7, 9, 12, 14, 15, 16, 17, 18, 21, 22, 24\}$
$Failure\text{-}inducing\ Chop(input3, argv[1], bitter@24) = \{7, 12, 16, 17, 18, 21, 22, 24\}$

$BwdSlice(input3, sweet@24) = \{7, 9, 12, 13, 24\}$
$Failure\text{-}inducing\ Chop(input3, argv[1], sweet@24) = \{7, 12, 13, 24\}$

backward slices on bitter, sweet and sour

$BwdSlice(input3, sour@24) = \{7, 9, 12, 14, 15, 16, 17, 18, 24\}$
$Failure\text{-}inducing\ Chop(input3, argv[1], sour@24) = \{7, 12, 16, 17, 18, 24\}$

Failure- Inducing chops algorithm :

1. use the delta debugging algorithm isolate *argv[1]* as the minimal failure-inducing input difference
2. Use dynamic slicing to locate fault code

for each of the faulty outputs contains the faulty statement 12

Dynamic Instrumentation

Divide a run into three phases, each phase has its unique instrumentation:

- **Phase One:**

  from the beginning of the execution to the predicate instance of interest, instrument a counter at a predicate. When it counts down to 0, it reached the interest predicate instance, enter phase 2.

- **Phase Two:**

  Instrument program switch the outcome of the interest predicate instance.

- **Phase Three:**

  Dynamic instrumenter – Valgrind cleans up all the instrumentation, complete the program run.

Limitations

- Cannot handle complex bugs in the program –
  require switching multiple predicate instances

- Cannot handle significant bugs –
  some functionality is missing from the program

Evaluation

Program	Found	Where	Which	False +ves
flex 2.5.319(a)	yes	gen.c @ 1813	0	0
flex 2.5.319(b)	no	search failed		
flex 2.5.319(c)	no	search failed		
grep 2.5	yes	grep.c @ 532	0	0
grep 2.5.1 (a)	yes	search.c @ 549	0	0
grep 2.5.1 (b)	no	search failed		
grep 2.5.1 (c)	yes	dfa.c @ 2854	2	0
make 3.80 (a)	yes	read.c @ 6162	143	1
make 3.80 (b)	yes	remake.c @ 652	1	0
bc-1.06	yes	storage.c @ 176	9	0
tar-1.13.25	yes	prepargs.c @ 81	0	0
tidy	yes	parser.c @ 3496	0	0
s-flex-v4	yes	flex.c @ 2978	0	0
s-flex-v5	no	search failed – error in DP		
s-flex-v6	no	search failed – error in DP		
s-flex-v7	yes	flex.c @ 9171	0	0
s-flex-v8	yes	flex.c @ 11833	0	0
s-flex-v9	yes	flex.c @ 5046	0	0
s-flex-v10	yes	flex.c @ 2687	1	0
s-flex-v11	yes	flex.c @ 3559	0	0

Successful/Failed Searches

Where:
file name + source line number at which the switched predicate can be found

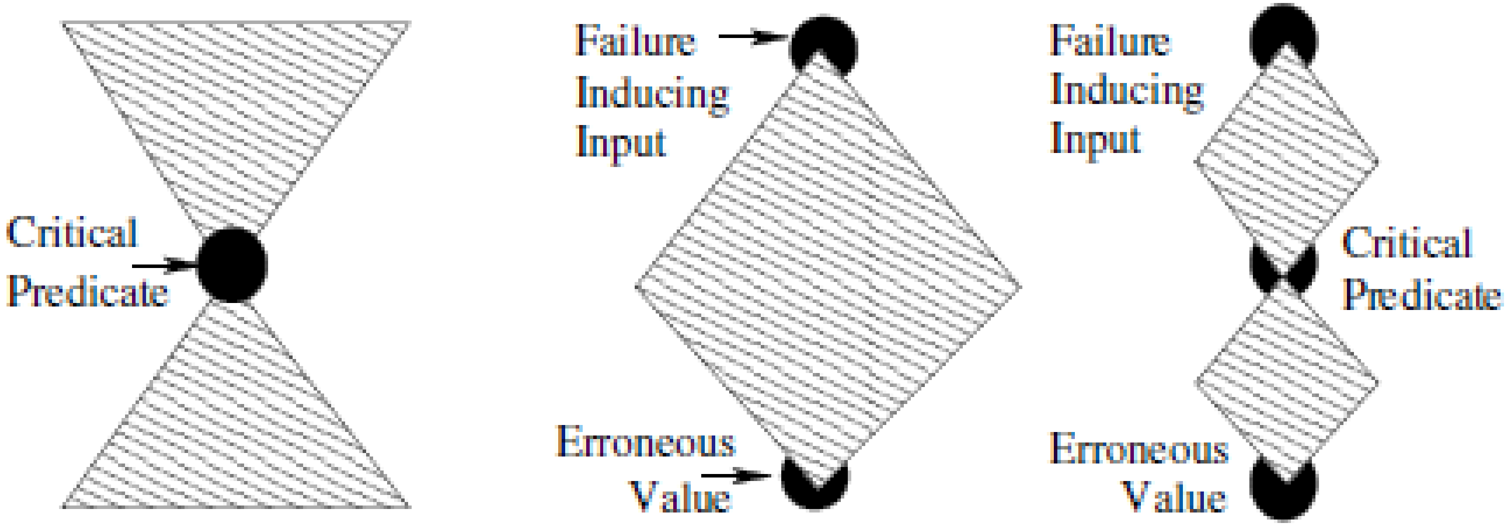
Which:
dynamic instance of the predicate that was switched

False +ves:
the number of dynamic predicate switches searched

Program	PRIOR
flex 2.5.319(a)	2.51 sec
flex 2.5.319(b)	search failed (364 min)
flex 2.5.319(c)	search failed (274 min)
grep 2.5	8.83 sec
grep 2.5.1 (a)	2.59 sec
grep 2.5.1 (b)	search failed (4 min 28 sec)
grep 2.5.1 (c)	4.46 sec
make 3.80 (a)	26.92 sec
make 3.80 (b)	30 min 37 sec
bc-1.06	0.49 sec
tar-1.13.25	2.83 sec
tidy	0.90 sec
s-flex-v4	8.76 sec
s-flex-v5	search failed (96 min 20 sec)
s-flex-v6	search failed (3 min 56 sec)
s-flex-v7	3.34 sec
s-flex-v8	34.35 sec
s-flex-v9	34.51 sec
s-flex-v10	2.76 sec
s-flex-v11	2.56 sec

Search time
taken by PRIOR
to locate the predicate instance switch

Locate *potentially faulty* code



(a) Bidirectional Slice (b) Failure–Inducing Chop (c) Bidirectional Chop

Bidirectional Slice:

- the critical predicate outcome was wrong due to incorrect values used in its computation. – *backward slice* of the critical predicate.
- changing the critical predicate outcome avoids the program crash – *forward slice* of the critical predicate captures the code causing the crash.

failure-inducing chop:

- *backward slice* of an incorrect output value
- *forward slice* of the failure-inducing input difference

Program	EXEC	BiS (%EXEC)	FiChop (%EXEC)	BiChop (%EXEC)	Where
flex 2.5.319(a)	1871	225 (12.03%)	256 (13.68%)	27 (1.44%)	Pred.
flex 2.5.319(b)	2198	-	102 (4.64%)	102 (4.64%)	-
flex 2.5.319(c)	2053	-	5 (0.24%)	5 (0.24%)	-
grep 2.5	1157	88 (7.61%)	731 (63.18%)	86 (7.43%)	Down
grep 2.5.1 (a)	509	111 (21.81%)	32 (6.29%)	25 (4.91%)	Down
grep 2.5.1 (b)	1123	-	599 (53.34%)	599 (53.34%)	-
grep 2.5.1 (c)	1338	453 (33.86%)	12 (0.90%)	12 (0.90%)	Up
make 3.80 (a)	2277	1372 (60.25%)	739 (32.45%)	739 (32.45%)	Up
make 3.80 (b)	2740	1436 (52.41%)	1104 (40.29%)	1051 (38.36%)	Up
bc-1.06	636	267 (41.98%)	102 (16.03%)	102 (16.03%)	Up
tar-1.13.25	445	117 (26.29%)	103 (23.15%)	45 (10.11%)	Down
tidy	1519	541 (35.62%)	164 (10.80%)	161 (10.60%)	Up
s-flex-v4	1631	37 (2.27%)	7 (0.43%)	7 (0.43%)	Pred.
s-flex-v5	1882	-	544 (28.91%)	544 (28.91%)	-
s-flex-v6	424	-	156 (36.79%)	156 (36.79%)	-
s-flex-v7	2045	836 (40.88%)	63 (3.08%)	63 (3.08%)	Up
s-flex-v8	610	280 (45.90%)	-	280 (45.90%)	Pred.
s-flex-v9	1396	230 (16.48%)	112 (8.02%)	112 (8.02%)	Pred.
s-flex-v10	1683	640 (38.03%)	574 (34.11%)	574 (34.11%)	Miss
s-flex-v11	1749	27 (1.54%)	102 (5.83%)	27 (1.54%)	Up

→ Location of fault code

Sizes of Bidirectional Slice / Failure-Inducing Chop / Bidirectional Chop

Conclusions

- Critical predicates be very often located in many real reported faulty programs
- They provide valuable clues to the cause of the failure and hence assist in fault location.

Questions ?

Thanks!