Semantic Mutation Analysis of Floating-Point Comparison

Haitao Dan and Robert M. Hierons

School of Information Systems, Computing and Mathematics, Brunel University

ICST, April 19, 2012

Contents

Issues of FPC

Not every real number can be represented by an FP number [1].

Non-deterministic behaviour for the same piece of source code with different configurations.

Calculation Error

```
double x, y, z;                                  1
...                                              2
z = x;                                           3
if(y > 0)                                        4
   x = x + y;                                    5
if(x <= z){                                      6
   // should not be here                         7
}                                                8
```

Considering $x = 1.0E30$ and $y = 5.0E13$
where $\epsilon = 1.407E14$

Non-deterministic behaviour

```
float a = 3.0, b = 7.0;                                  1
float c = a / b;                                         2
if (c == a / b)                                          3
   puts("Comparison succeeds");                          4
else                                                     5
   puts("Unexpected result");                            6
```

Traditional and Semantic Mutation Testing

	Traditional	**Semantic**
Idea	errors can be simulated by small syntactical changes.	misunderstandings errors can be simulated by semantic changes
Technique	generate mutants by applying mutation operators.	
Implication	high mutation score $\rightarrow$ T is likely to be more efficient in finding faults similar to the generated mutants.	high mutation score $\rightarrow$ T is likely to be more efficient in capturing semantic errors.

Differences between MT and SMT

- Possible fewer mutants, as SMT operators have more complex constrains.
- Consequently, semantic mutation operators can be harder to be implemented.

Example revisit

```
float a = 3.0, b = 7.0;                                  1
float c = a / b;                                         2
if (c == a / b)                                          3
   puts("Comparison succeeds");                          4
else                                                     5
   puts("Unexpected result");                            6
```

FPC Operator

Introducing a tolerance in comparing two FP numbers.

```
float a, b;                 float a, b;
...                         ...
if(a==b){          →        if(flpcmp(a,``=='',b)){
...                         ...
}                           }
```

Three Algorithms and Six FPC Operators

There are 6 operators: three for *float* numbers (MFC_C, MFC_F and MFC_H) and three for *double* numbers (MDC_C, MDC_F and MDC_H).

- Constant tolerances: *FLT_EPSILON* or *DBL_EPSILON*.
- Scale constant tolerances according to the value of operands, first proposed by Knuth [2].
- A hybrid approach that combines the two algorithms.

Research Questions

1. Do programmers use FPCs in C programs?
2. Is it easy to kill FPC mutants? If it is not easy, can we find a technique to generate test cases that kill FPC mutants?
3. What is the impact of using different FPC algorithms in terms of SMT?
4. Do FPCs behave consistently on different computers with different configurations (portability of FPCs)?

FPCs in Programs

Program	LoC	NFPC	ANF	Description
daimonin_client	32534	10	0.0307	2D game
bitlbee	44264	0	0	chatting
dbmail	27585	0	0	email server
expat	13999	0	0	XML parser
gstreamer	225036	161	0.0715	media library
iodine	7486	0	0	DNS tunnel
lasso	46568	0	0	id management
libtinymail	116074	0	0	email lib
libxcb	18666	0	0	X protocol
Total	532212	171	0.0321	
R	249718	3289	1.37	statistics and graphics
nmath	9255	1605	17.3	math module of R

Discussion ...

FPC was not often used in general C programs, but for some mathematics libraries such as *nmath* in *R*, FPCs were intensively used.

Test Suites

Function	$\mathbb{R}$		$\mathbb{M}$	
	Number	COV[1]	Number	COV
bd0	1000	0.938	3	0.936
dbinom_raw	1000	0.556	6	0.556
dpois_raw	1000	0.750	4	0.750
gamma_cody	1000	0.906	3	0.415
lgammacor	1000	1.000	3	0.889
pgamma_raw	1000	0.923	5	0.731
pnbeta_raw	1000	0.919	11	0.892
pnchisq_raw	1000	0.708	2	0.215
qchisq_appr	1000	0.963	3	0.597
stirlerr	1000	0.857	4	0.929
bratio	1000	0.450	16	0.641
Total/Average	11000	0.815	60	0.686

[1] Statement coverage.

Semantic Mutation Analysis of FPCs

Mutants		Test Suite $\mathbb{M}$				Test Suite $\mathbb{R}$			
Name	NUM[2]	KM[3]	MS[4]	KTC[5]	SS[6]	KM	MS	KTC	SS
MFC_C	75	24	0.320	28	0.168	2	0.027	666	0.333
MFC_F	75	25	0.333	27	0.167	1	0.013	666	0.666
MFC_H	75	24	0.320	28	0.168	2	0.027	679	0.340
MFC_*	225	73	0.324		0.168	5	0.022		0.402
MDC_C	111	33	0.297	40	0.196	1	0.009	133	0.133
MDC_F	111	19	0.171	20	0.235	0	0.000	0	0.000
MDC_H	111	36	0.324	43	0.202	1	0.009	130	0.130
MDC_*	333	88	0.264		0.208	2	0.006		0.132
All	558	161	0.289		0.190	7	0.013		0.325

[2]Number of the mutants generated.

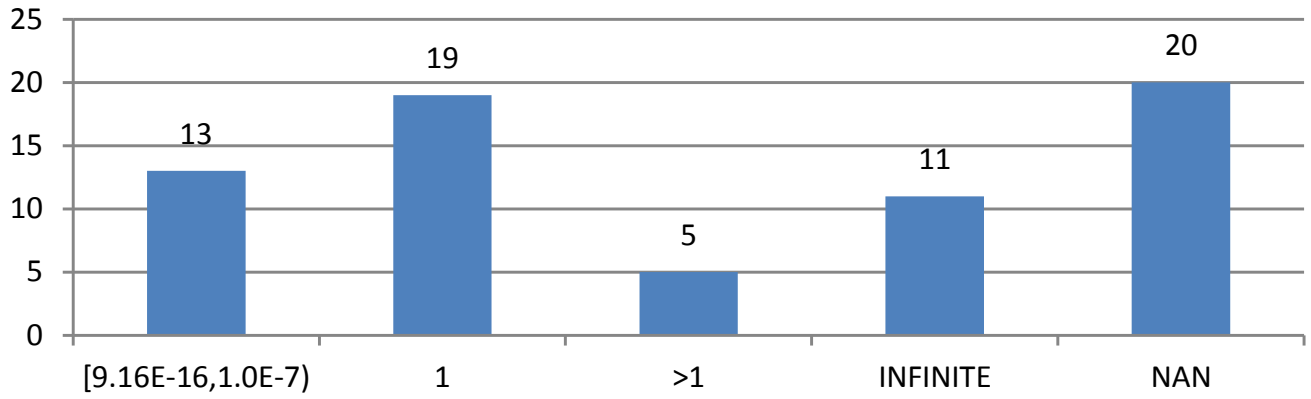
[3]The number of killed mutants.

[4]Mutation score.

[5]The total number of test cases that kill this type of mutants.

[6]The average semantic size of the killed mutants.

Output differences of killed mutants



Discussion ...

Mutants generated by FPC operators were hard to kill using random test suites. Whereas, the manual test suite achieved a significant higher average mutation score.

Comparison of Different Types of Tolerance

$\mathbb{M}$ \ $\mathbb{M}$	*_C	*_F	*_H
*_C	57		
*_F	31	44	
*_H	55	36	60

(1)

$\mathbb{R}$ \ $\mathbb{R}$	*_C	*_F	*_H
*_C	3		
*_F	1	1	
*_H	3	1	3

(2)

$\mathbb{M}$ \ $\mathbb{R}$	*_C	*_F	*_H
*_C	0		
*_F	1	1	
*_H	0	0	0

(3)

Discussion ...

For the given test suites, mutants generated using different algorithms cannot replace each other.

Porting FPCs between Computers[7]

- From *Computer 1* to *Computer 2*, manual test suite kills 3 more *_C, 1 more *_H mutants, and 1 less *_F mutants.
- For the random test suite, there is 1 mutant that is only killed on *Computer 1*.

[7] *Computer 1*: a 32 bit Linux OS with an Intel processor. *Computer 2*: a 64 bit Linux OS with an AMD processor.

Discussion ...

It is shown that FPC does have portability problems.

Conclusions and Future Work

- FPC mutants are generally hard to kill.
- The SMT-based analysis categorises FPC mutants. The mutants cannot killed by the manual suite may imply that the corresponding FPC is "benign". Software engineers may need to pay more attentions to mutants that can be killed.
- Analyse equivalent mutants generated by FPC operators.
- Other problems in FP arithmetic, for example type casting.
- Automate the killing of FPC mutants.

Thank You!
and
Questions?

References

D. Goldberg.
What every computer scientist should know about floating-point arithmetic.
ACM Computing Surveys, 23(1):5–48, 1991.

D.E. Knuth.
The art of computer programming: Seminumerical algorithms, volume 2.
Addison-Wesley, 1981.