An Intuitive Approach to Determine Test Adequacy in Safety-critical Software

P. Arun Babu, C. Senthil Kumar, N. Murali, T. Jayakumar

ACM SIGSOFT Software Engineering, Vol. 37, 2012

2013. 3. 27.

Presented by Kim, Hanseok

KAIST SE LAB

KAIST Software Engineering Laboratory

Contents

Introduction (1/2)

- What is software reliability?
  - It is the probability of failure-free software operation for a specified period of time in a specified environment [1]
  - High reliability is essential for the safety-critical system
- How to demonstrate the software meets a required reliability?
  - Formal methods
    - Prove that the software meets its functional requirements
    - Not always feasible to ensure complete software verification
  - Software testing
    - The most popular methodology to establish the reliability
    - Not possible to test for every possibility

1) ANSI/IEEE, 1991, 1995

Introduction (2/2)

- Motivation
  - Safety and mission critical software is expected to be thoroughly tested
    - Programmers and testers must write good test cases which can verify the behavior of the entire system
  - Exhaustive testing of the software is usually impractical
    - The two main challenges are generation of effective test cases and demonstration of testing adequacy
- Goal of this paper
  - Propose an intuitive and conservative approach to determine the test adequacy of safety-critical software
    - Combine conservative test coverage and mutation testing

Overview of our approach

- Test adequacy with case study (CTMS)

Step 1 (Test case generation and coverage)

- Test case generation
- Conservative test coverage

Step 2 (Mutation Properties)

- Effectiveness of mutation testing
  - Mutation operator
- Confidence of test cases
  - Static, dynamic analysis and PCA

Step 3 (Mutation Score)

- Mutation score
- Equivalent mutants detecting technique

Step 4 (Test Adequacy)

- Test coverage (with weightage)
- Test adequacy (combination test coverage + mutation score)

System description

- Core Temperature Monitoring System
  - Keep track of the core temperature of a nuclear reactor through thermocouples
  - Detect anomalies such as plugging of fuel sub-assemblies, error in core loading and so on
  - Has 4 major modules
    - H/W interface module, N/W interface module
    - Diagnostic module, Main module (performs the actual safety-critical function)
  - Passed following static, dynamic and security checkers
    - No warnings with static analyzers; Clang, Cppcheck
    - No warnings with dynamic analyzers; Valgrind, Electirc-Fence
    - No vulnerabilities with security scanners; FlawFinder, RATS, …

    * These tools are used to investigate mutant properties

Test case generation

- Problems
  - Safety-critical s/w is often expected to have 100% MC/DC and LCSAJ coverage [1, 2]
    - For the coverage, we need large number of test cases
    - Large number of test cases are often difficult to hand code and may require automatic test case generation

- Our system (CTMS)
  - Around 300,000 test cases were generated
    - Used pseudo and true random number generation
  - 11,436 unique execution path test cases were selected
    - Removed redundant test cases which follow the same path of execution (by MD5 hash of the coverage information)

1) Hayhurst, K. et al , A Practical Tutorial on Modified Condition/Decision Coverage, 2001
2) Woodward, M.R. et al, Experience with Path Analysis and Testing of Programs, 1980

Test coverage

- Coverage metric
  - As single control coverage could be misleading, we define this metric, defined by
    - minimum of LCSAJ, MC/DC coverage, branch, statement coverage
  - As the branch coverage is always less than or equal LCSAJ coverage, the conservative test coverage is defined as:
    - *min(LCSAJ coverage, MC/DC, Statement coverage)*

Mutation properties (1/5)

- For effectiveness of mutation testing
  - Effectiveness of mutation testing may be judged by the quality and number of mutation operators used
    - Used 60 mutation operators

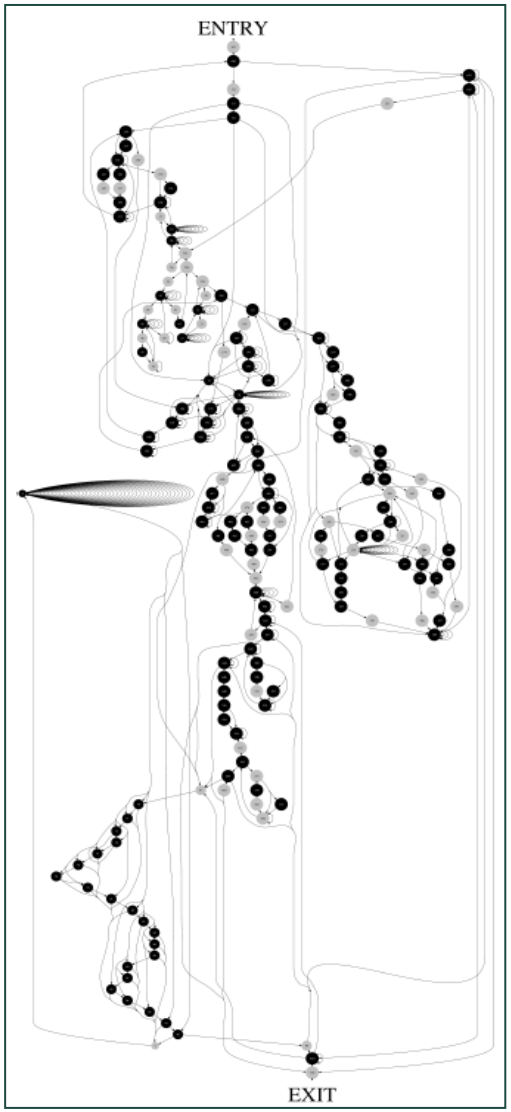
#	Substring	Mutated to *(separated by ,)*
1	<	!=, >, <=, >=, ==
2	>	!=, <, <=, >=, ==
3	<=	!=, <, >, >=, ==
4	>=	!=, <, <=, >, ==
5	==	!=, <, >, <=, >=
6	!=	==, <, >, <=, >=
7	=	= -, == , = 0 *, = ~ , = !
8	=	= 0; //, = NULL; //
9	+	-, *, /, %
10	-	+, *, /, %
11	*	+, -, /, %
12	/	%, *, +, -

...

    - Generated 2,013 mutants
      - 1,670 (82.96%) mutants were killed
      - 324 (16,09%) mutants could not be killed
      - 19 (0.94%) mutants made segmentation faults/bus errors

Mutation properties (2/5)

- For confidence on mutation testing
  - Fault must be induced at all possible execution paths
    - All paths are concatenated all the LCSAJs (= Faults induced at various jump points of LCSAJs)
      - Fault is indicated by dark colored nodes
    - Figure indicates that there exists no path where faults have not been induced and caught
      - Giving confidence on the effectiveness of mutation testing



Mutation properties (3/5)

- For confidence on the test cases
  - Need to understand
    - The characteristics of unkilled mutants
    - The differences from the killed mutants
  - Static analysis (using Splint, Clang)
    - Not clearly differentiate between killed and unkilled mutants

Static analysis

Using Splint						Using Clang					
Killed mutants			Unkilled mutants			Killed Mutants			Unkilled mutants		
No. of mutants		Warnings	No. of mutants		Warnings	No. of mutants		Warnings	No. of mutants		Warnings
861	(50.97 %)	0	142	(43.82 %)	1	1586	(93.9 %)	0	304	(93.82%)	0
584	(34.57 %)	1	129	(39.81 %)	0	91	(5.38 %)	1	19	(5.86 %)	1
236	(13.97 %)	2	35	(10.8 %)	2	11	(< 1 %)	2	1	(< 1 %)	2
5	(< 1 %)	3	6	(1.85 %)	3	1	(< 1 %)	3			
1	(< 1 %)	4	3	(< 1 %)	22						
1	(< 1 %)	5	2	(< 1 %)	16						
1	(< 1 %)	19	2	(< 1 %)	7						
			2	(< 1 %)	4						
			1	(< 1 %)	9						
			1	(< 1 %)	8						
			1	(< 1 %)	5						

Mutation properties (4/3)

- For confidence on the test cases (Cont'd)
  - Dynamic analysis (using Valgrind, Electric-Fence)
    - None of unkilled mutants are likely to have any memory corruptions or leaks

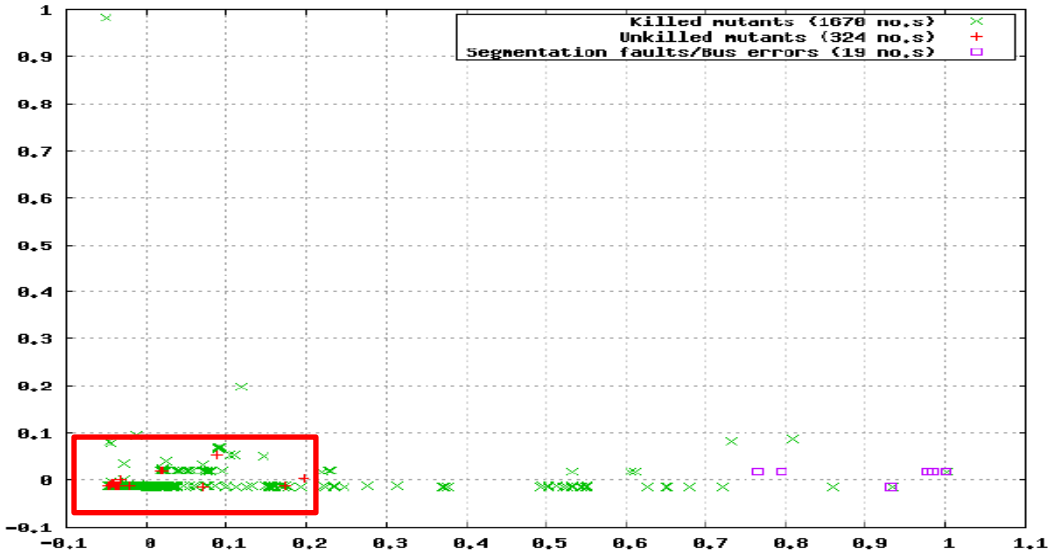
Using Valgrind and Electric-Fence

Killed mutants			Unkilled mutants		
No. of mutants		Errors	No. of mutants		Errors
1604	(94.96 %)	0	324	(100%)	0
37	(2.19 %)	1			
20	(1.18 %)	234024			
6	(< 1 %)	34287			
6	(< 1 %)	11666			
2	(< 1 %)	11152			
2	(< 1 %)	2			
(others) 12	(< 1 %)	7216.33 (Avg)			

    - 96.6% of unkilled mutants don't have any change in the test coverage
      - 96.6% (coverage impact) is calculated using "gcov" program
      - Number of times a statement/branch/jump/function-call was executed in a mutant with respect to the original program

Mutation properties (5/5)

- For confidence on the test cases (Cont'd)
  - Result of principal component analysis (PCA)
    - The unkilled mutants have little variance when compared to the killed mutants
      - Similar static, dynamic, and coverage properties
    - Little confidence that majority of the unkilled mutants are likely to be equivalent

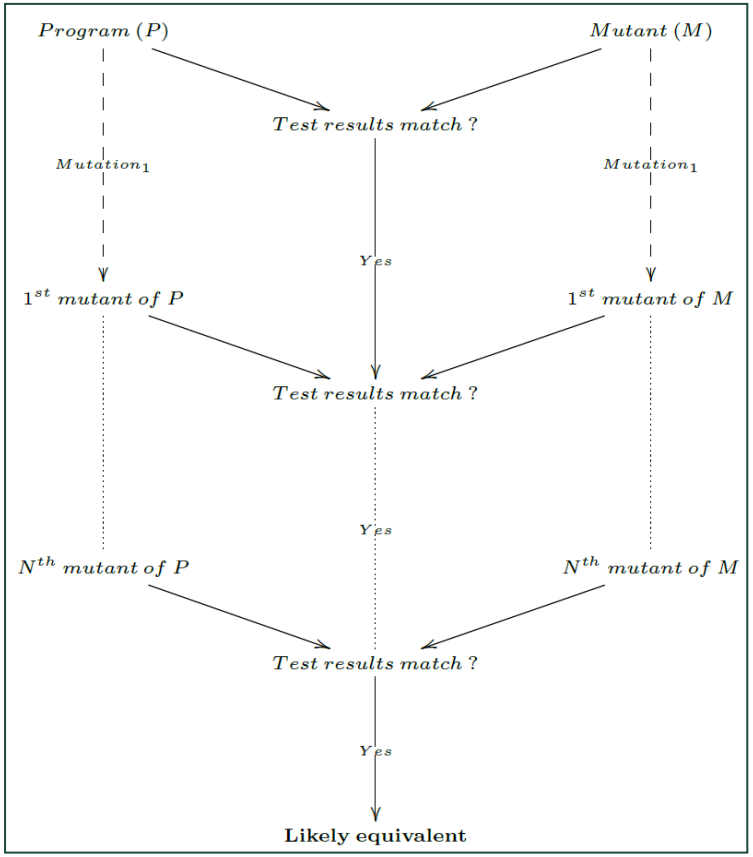


Mutation score (1/2)

- Mutation score
  - A method or strategy to check the effectiveness or accuracy of a testing program
  - The result of mutation testing, defined as:
    - Mutation score = K / (G – E)
      - K : # of mutants killed, G : # of mutants generated,
      - E : # of equivalent mutants
  - Our system (CTMS)
    - 2,013 generated mutants
    - 1,670 (≈ 83%) killed mutants
    - 324 (≈ 17%) unkilled mutants
    - How to detect the equivalent mutants among the unkilled mutants?

Mutation score (2/2)

- Mutation score (Cont'd)
  - Detection algorithm
    - If P is Program, M is its equivalent mutant created by injecting a fault F in the statement S,
    - Then P'(mutant of P) and M'(mutant of M), created by injecting faults F' in succeeding S, must also be equivalent
    - <u>If many such equivalent P' and M' are generated, then P and M are likely to be equivalent</u>



  - Our system (CTMS)
    - 11 mutants among 324 unkilled mutants are non-equivalent
    - Mutation Score = 1670/(2013-(324-11)) = 1670/1700 ≈ 1

Test Adequacy (1/2)

- Test coverage
  - The degree to which the source code of a program has been tested
  - Give weightage to large, complex, and frequently called function
  - Test coverage of CTMS is calculated as a weighted average, given as:

$$Test\ coverage\ = \frac{\sum t_i\, w_i}{\sum w_i}$$

$$w_i = No.of\,statements \times Cyclomaticcomplexity \times Call\,frequency$$

$$t_i = min\ (LCSAJ\ coverage,\ MC/DC,\ Statement\ coverage)$$

Test Adequacy (2/2)

- Adequacy of testing
  - The rigor of software testing expressed as the adequacy of testing, is given by:

$$
\begin{aligned}
Adequacy\ of\ testing &= Mutation\ score \times Test\ coverage\ \% \\
Adequacy\ of\ testing &\approx 1 \times 98.8\ \% \\
Adequacy\ of\ testing &\approx 98.8\ \%
\end{aligned}
$$

  - 98.8% indicate that enough software testing has been carried out for the CTMS software
  - In safety critical industries, the result may serve as one of the <u>inputs for the initial software reliability estimate</u> before permitting the software to be used in the field

Conclusion

- Contribution
  - Demonstrate an approach to determine the test adequacy thorough a safety-critical case study
    - Combination conservative test coverage with mutation score
  - Investigate characteristic of unkilled mutants
    - Similar static, dynamic and coverage properties (by PCA)
    - Probably equivalent mutants
  - Demonstrate a technique to identify equivalent mutants
    - Use as input parameter of mutation score
- Future work
  - Implementation of the proposed approach on various safety related software

Discussion

- Pros
  - Supporting the approach through a safety-critical case study in a nuclear reactor
- Cons
  - Too intuitive approach
  - Need for more case-study and more concrete validation

Thank You .

KAIST SE LAB
KAIST Software Engineering Laboratory

About Authors

- P. Arun Babu
  - Works as IGCAR
    (Indira Gandhi Centre for Atomic Research)
  - Interested in
    - Software reliability, test adequacy (2012~)
    - Formal methods (2010~2011)

- C. Senthil Kumar
  - Works as Safety Research Institute, AERB
    (Atomic Energy Regulatory Board)

Appendix

- Coverage Type
  - Condition Coverage
    - The possible outcomes of (“True” or “False”) for each condition are tested at least once
  - Decision Coverage (Branch Coverage)
    - The possible outcomes of the decision are tested at least once
  - Condition Decision Coverage
    - The possible outcomes of each condition and of the decision are tested at least once

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

<Condition Coverage>

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

<Decision Coverage>

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

<Condition/Decision Coverage>

Appendix

- Coverage Type (Cont'd)
  - <u>Modified Condition Decision Coverage (MC/DC)</u>
    - Every possible outcome of a condition determines at least once the outcome of the decision

A	B	A OR B
1	1	1
1	0	1
0	1	1
0	0	0

<Modified Condition/Decision Coverage>

Condition A (True) -> Decision (True)
Condition A (False) -> Decision (False)
Condition B (True) -> Decision (True)
Condition B (False) -> Decision (False)

- Multiple Condition Coverage needs 2^n test situation
- But, MC/DC needs only n+1 test situation
  - n : # of condition

Appendix

- LCSAJ coverage
  - LCSAJ (Linear Code Sequence And Jump)
    - A program unit composed of a textual code sequence that terminates in a jump to the beginning of another code sequence and jump
    - LCSAJ (X,Y,Z) where X and Y are, respectively, the locations of the first and the last statements and Z is the location to which the statement at Y jumps
  - LCSAJ Coverage = I/L
    - I = # of LCSAJs exercised at least once
    - L = Total # of LCSAJs

Appendix

LCSAJ coverage (Cont'd)

- LCSAJ Example

```
1   begin
2       int x, y, p;
3       input (x, y);
4       p = 1;
5       count=y;
6       while(count>0) {
7           p=p*x;
8           count=count-1;
9       }
10      output(p);
11  end
```

LCSAJ	Start Line	End Line	Jump to
1	1	9	6
2	6	9	6
3	6	6	10
4	1	6	10
5	10	11	exit

T = {t_1: <x=5, y=0>, t_2: <x=5, y=2>}

t_1 covers (1,6,10), (6,6,10) and (10,11,exit)
t_2 covers (1,9,6), (6,9,6), (6,6,10) and (10,11,exit)

T covers all 5 LCSAJs
=> LCSAJ coverage = 5/5 = 1

Appendix

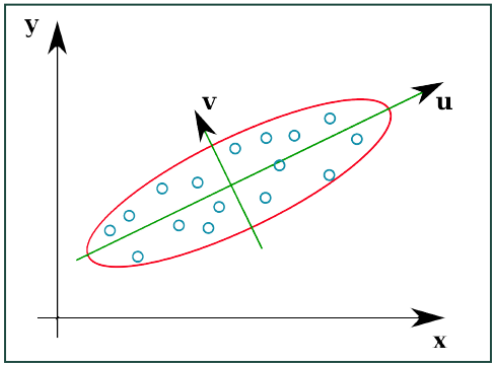
- Mutation testing
  - A method of inserting faults into programs to test whether the tests pick them up, thereby validating or invalidating the tests
  - It is done by selecting a set of mutation operators and then applying them to the source program
    - Mutation operators : Operators for inserting faults
    - ex) replace “&&” with “| |”
  - The result of applying one mutation operator to the program : Mutant

Appendix

- Killed mutants
  - If test cases detect differences(=fail) in mutants, then the mutants are said to be killed
- Equivalent mutants
  - Always acts in the same behavior as the original program (functionally identical)
  - When we can't find a test case that could kill the mutant, the result program is equivalent to the original one
- Unkilled mutants (Live mutant)
  - If 1) a mutant is equivalent to the original program or 2) the test set is inadequate to kill the mutant, then the mutant is said to be unkilled (live)

Appendix

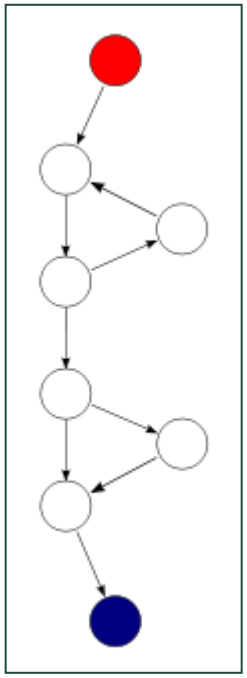
- Principal component analysis (PCA)
  - Definition
    - A statistical procedure that reduces data dimensionality by performing a covariance analysis between factors



  - Goal
    - Extract the most important information from the data table
    - Compress the size of the data set by keeping only this important information
    - Simplify the description of the data set
    - Analyze the structure of the observations and the variables

Appendix

- Cyclomatic complexity
  - Complexity of a program(CFG, Control Flow Graph)
    - M = E – N + 2P
      where E(# of edges), N(# of nodes),
      P(# of connected components)



» E = 9, N = 8 and P = 1

» Complexity M = 9 - 8 + 2 * 1 = 3