Better Predicate Testing

Gary Kaminski, Paul Ammann, Jeff Offutt

Software Engineering

George Mason University

Fairfax, VA USA

www.cs.gmu.edu/~offutt/

offutt@gmu.edu

Covering Logic Expressions

- Logic expressions show up in many situations
- Covering logic expressions is required by the US Federal Aviation Administration for safety critical software
- Logical expressions can come from many sources
  - Decisions in programs
  - UML : FSMs and statecharts, activity diagrams
  - Requirements
  - SQL queries
- Tests are a subset of expressions' truth assignments

Logic Predicates and Clauses

- A *predicate* is an expression that evaluates to a boolean value
- Predicates can contain
  - boolean variables
  - non-boolean variables that contain >, <, ==, >=, <=, !=
  - boolean function calls
- Internal structure is created by logical operators
  - $\neg$ – the *negation* operator
  - $\wedge$ – the *and* operator
  - $\vee$ – the *or* operator
  - $\rightarrow$ – the *implication* operator
  - $\oplus$ – the *exclusive or* operator
  - $\leftrightarrow$ – the *equivalence* operator
- A *clause* is a predicate with no logical operators

Power of Logic Testing

- Logic expressions encode the behavior of software
- Logic expressions define the domain of values for which the software behaves in a certain way
- Logic expressions are often
  - Complicated
  - Subtle
  - Easy to get wrong, both in design and implementation

Testing logic predicates is a cost-effective way to find many subtle software faults

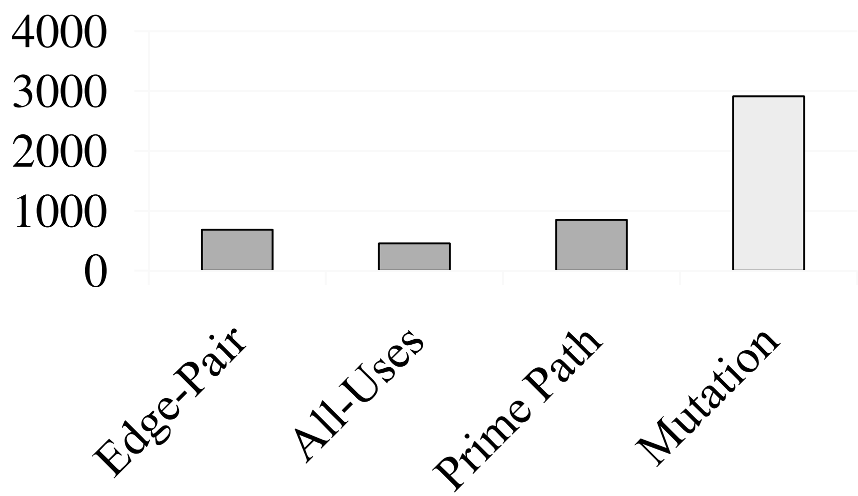
Problems Addressed

- This theoretical talk presents results on two problems with logic predicate testing :
  1. Redundant mutation operators for predicate testing
  2. Weakness of major logic testing criterion : MCDC
- Solution based on theoretical analysis
- Solution can be immediately used to create better tools and stronger criteria, with very slight cost

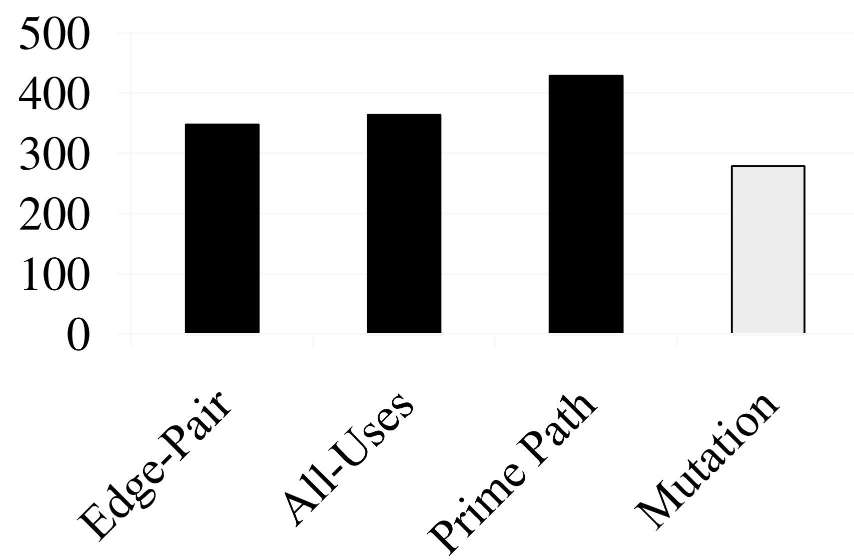
(1) Redundancy in Mutation

- Mutation is widely considered to be "expensive"
- This expense is largely based on the high number of test requirements—mutants
- But Li et al. found that mutation needed fewer tests !

Number of Test Requirements 29 Java Classes



Number of Tests 29 Java Classes



Li, Praphamontripong, Offutt, An experimental comparison of four unit test criteria, Mutation 2009

Eliminating Redundancy

- This is strong evidence that mutation tools use many redundant operators
- A more clever mutation system should have less redundancy
- Fewer mutants means less work for the tester … cheaper!

Mutation Predicate Testing

- Traditional ROR operator :

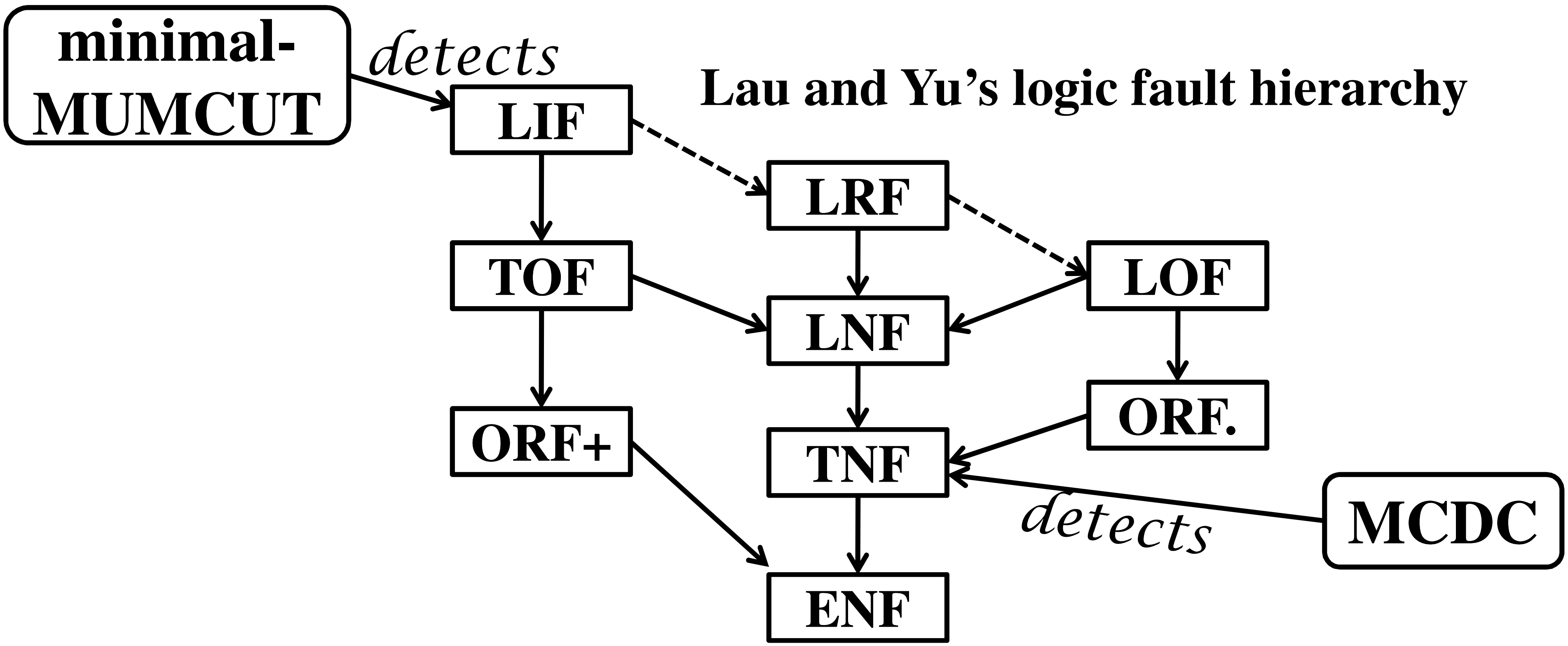
Each occurrence of a relational operator (<, >, <=, >=, =, !=) is replaced by each other operator, and the expression is replaced by *True* and *False*.

- **Example:**
  - **a > b**
  - **M1: a < b**
  - **M2: a <= b**
  - **M3: a >= b**
  - **M4: a == b**
  - **M5: a != b**
  - **M6: *true***
  - **M7: *false***

Mutation Predicate Testing

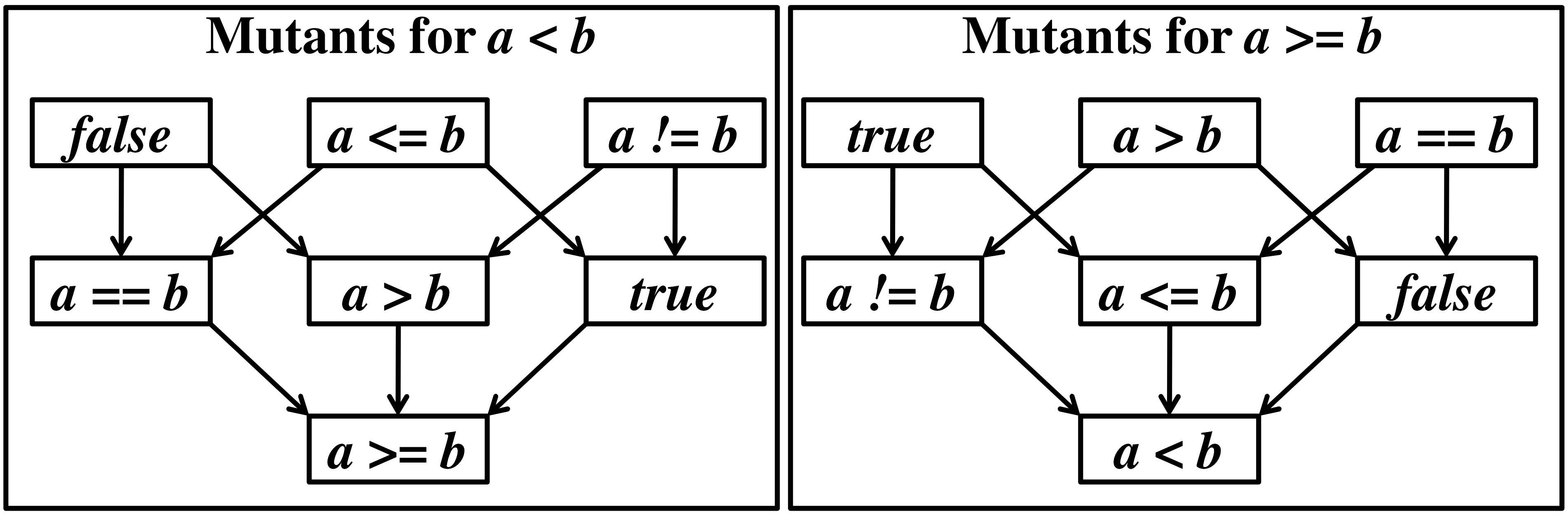
A *fault hierarchy* establishes theoretical dominance relations among faults:

If fault A *dominates* fault B, then any test that detects fault A will by definition detect fault B



ROR Mutant Hierarchy

If *mutant* A dominates *mutant* B, then any test that detects *mutant* A will by definition detect *mutant* B



A Cheaper ROR Operator

Each occurrence of a relational operator (<, >, <=, >=, =, !=) is replaced by operators as follows:

- **< : <=, !=, False**
- **> : >=, !=, False**
- **<= : <, ==, True**
- **>= : >, ==, True**
- **== : <=, >=, False**
- **!= : <, >, True**

Saves four mutants for each relational operator !

(2) Weakness of MCDC

- MCDC was invented in the early 1990s
- Research community has invented many additional logic criteria since
  - MCDC is weaker than MUMCUT (& Minimal-MUMCUT)
  - MCDC is weaker than ROR-mutation
- MCDC works at the clause level
- ROR works at the relational operator level

Solution : Extend MCDC to the relational operator level

Stronger MCDC

- MCDC can be extended to include requirements to kill ROR mutants
- Method :
  - MCDC requires clause $c = x \ op \ y$ to have two values *True* and *False*
  - Cheaper-ROR requires c to have three values :

    $x < y, x == y, x > y$
  - The two MCDC values will always satisfy at least two of the cheaper-ROR requirements
  - Add one additional test to cover the third

Cost is Minor

- MCDC on a predicate with *N* clauses requires *N+1* .. *2N* tests
- MCDC + ROR requires *N* more (*2N+1* .. *3N* tests)
- Algorithm and proof in paper

Example

$p = a \wedge b \vee c$

a = (a1 < a2), b = (b1 <= b2), c = (c1 == c2)

The following test set satisfies MCDC :

T = { t1, t2, t3, t4} = { ttf, tft, tff, ftf }

Which can be refined with the following value assignments :

	Test	Value	a1	a2	b1	b2	c1	c2	a	b	c
	t1	TTF	5	6	10	11	21	22	<	<	
	t2	TFT	5	6	11	10	21	21			==
	t3	TFF	5	6	11	10	21	22		>	<
	t4	FTF	6	5	10	11	21	22	>		
ROR tests	New	(t1)	**5**	**5**	10	11	21	22	==		
	New	(t1)	5	6	**10**	**10**	21	22		==	
	New	(t2)	5	6	11	10	**22**	**21**			>

Recommendations

1. Mutation tools
   – Future mutation tools should use cheaper-ROR
   – No loss in strength
   – Savings of four test requirements (mutants) for each relational operator
2. Logic criteria
   – Extend MCDC to MCDC + ROR
   – Better: Replace MCDC with Minimal-MUMCUT + ROR
   – Logic testing should apply to the relational operator level
   – Small increase in the number of tests
   – Large increase in the testing strength

Summary

- RTCA-DO-178B has been in effect for almost 20 years
- MCDC was a brilliant idea
- But recent advances have led to better logic criteria
- We continue to reduce the cost of applying mutation in practice

Future Work

1. Empirical evidence for the increased ability of MCDC-RORG to find faults

2. Can we apply similar analysis to reduce the number of mutants from other operators?
   – Arithmetic operators ?
   – Variable replacement ?

Contact

Jeff Offutt

offutt@gmu.edu

http://cs.gmu.edu/~offutt/