Automating the Generation of Mutation-based Test Cases

Mike Papadakis

Department of Informatics
Athens University of Economics and Business

Mutation Testing

- **White-box, fault-based testing technique**
  - *Considered as one of the most effective techniques at detecting faults*
- **Produces faulty program versions**
- **Aim: find test cases that distinguish the original from fault-mutant program versions**
- **Assessing test adequacy**
  - Dead mutant ratio *(Distinguished outputs)*

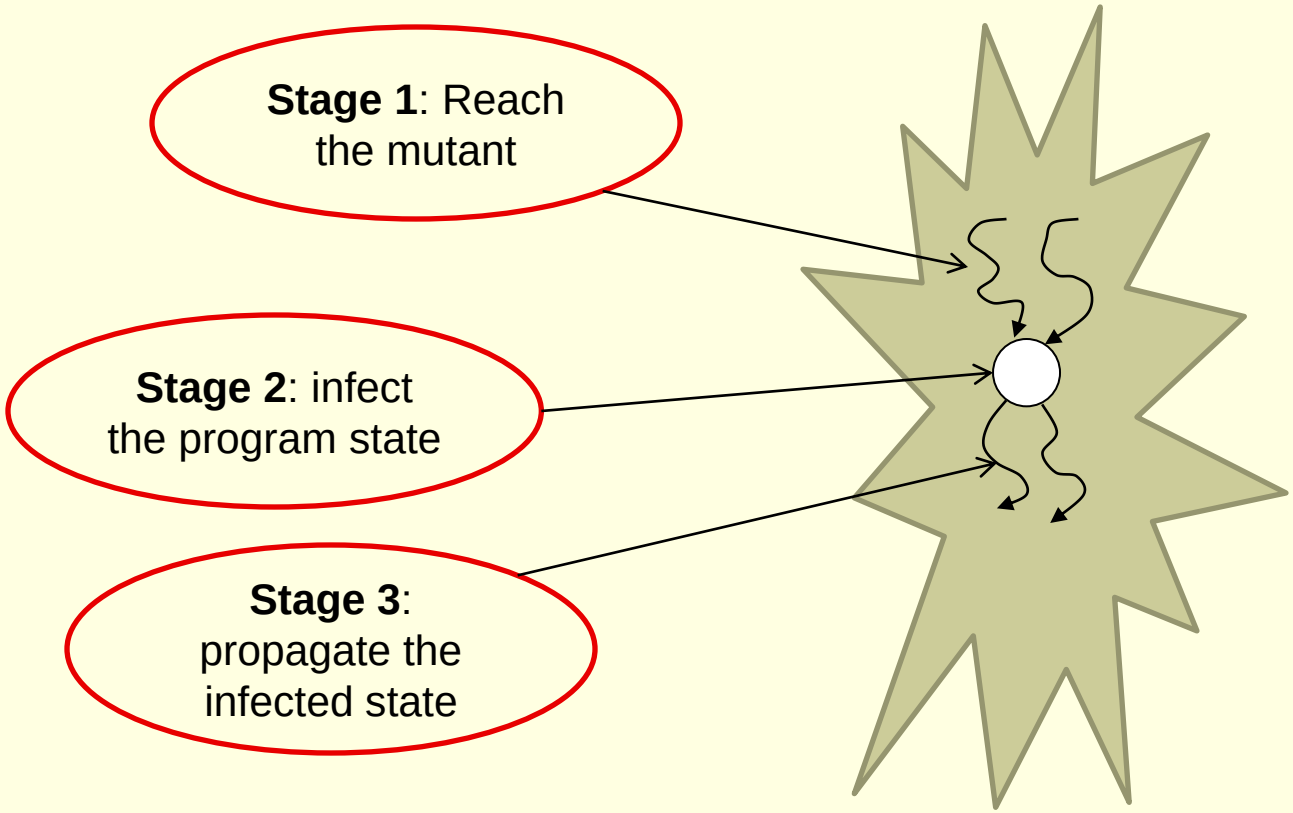
$$\text{MutationScore} = \frac{\text{No. of Dead Mutants}}{\text{No. of Mutants - No. of Equivalent Mutants}}$$

Search Based Testing

- **Automatic generation of test cases**
  - *Test cases able to kill the introduced mutants*
- **Formulate test generation to a search program**
  - *Use of search based optimization techniques*
  - *Dynamic program execution*
  - *Fitness function*
- **Hill Climbing (*AVM*)**
  - *Quite effective in structural testing*
  - *Repeatedly adjusts the program inputs*

Killing Mutants

- In order to kill a mutant, tests must



- Joint satisfaction *(Reach && Infect && Propagate)*

Killing Mutants (*Search Based*)

- **Measure the closeness of <u>reaching a mutant</u>**
- **Measure <u>mutation distance</u>**
  - *Closeness of weakly killing the targeted mutant. Mutant necessity condition:* ***Original expression ≠ Mutated expression***
  - *Use simplified necessity fitness for improved performance*
- **Measure <u>Predicate mutation distance</u>**
  - *Closeness of* making changes on the mutant and original program predicates
- **Approximate <u>Sufficiency condition</u>**
  - *Measure the closeness of reaching specific program nodes (likely to expose mutants)*

Fitness function (*reaching a mutant*)

- approach level
  - *The number of control dependent nodes missed*
- branch distance

Expression	True Branch	False Branch
a == b	abs(a - b)	a == b?k : 0
a != b	a != b? 0 : k	abs (a != b?a - b : 0)
a < b	abs (a < b?0 : a - b + k)	abs (a < b?a - b + k : 0)
a <= b	abs (a <= b?0 : a - b)	abs (a <= b?a - b : 0)
a > b	abs (a > b?0 : a - b + k)	abs (a > b?a - b + k : 0)
a >= b	abs (a >= b?0 : a - b)	abs (a >= b?a - b : 0)
a || b	min[fit(a), fit(b)]	fit(a) + fit(b)
a && b	fit(a) + fit(b)	min[fit(a), fit(b)]

Fitness function (*mutation distance*)

Operator	Original expression	Mutant Fitness	
Relational	a > b	a >= b: abs(a-b) a < b: k a <= b: 0	a != b: abs(a-b+k) a == b: abs(a-b) true: abs(a-b) false: abs(a-b+k)
	a >= b	a > b: abs(a-b) a < b: 0 a <= b: k	a != b: abs(a-b) a == b: abs(a-b+k) true: abs(a-b+k) false: abs(a-b)
	a < b	a > b: k a >= b: 0 a <= b: abs(a-b)	a != b: abs(a-b+k) a == b: abs(a-b) true: abs(a-b) false: abs(a-b+k)
	a <= b	a > b: 0 a >= b: k a < b: abs(a-b)	a != b: abs(a-b) a == b: abs(a-b+k) true: abs(a-b+k) false: abs(a-b)
	a != b	a > b: abs(a-b+k) a >= b: abs(a-b) a < b: abs(a-b+k) a <= b: abs(a-b)	a == b: 0 true: abs(a-b) false: k
	a == b	a > b: abs(a - b) a >= b:abs(a-b+k) a < b: abs(a - b) a <= b:abs(a-b+k)	a != b: 0 true: k false: abs(a-b)

Fitness function (*mutation distance*)

- **Example**

  (a > b) ≠ (a >= b)

  If (a == b) then

  a > b -> false

  a >= b -> true

  else

  (a > b) == (a >= b)

- **Mutation distance**

  abs(a - b)

Fitness function (*mutation distance*)

Operator	Original expression	Mutant Fitness	
Arithmetic	a + b	a - b:k a * b:k a / b:k	a % b:k a:k b:k
	a – b	a + b:k a * b:k a / b:k	a % b:k a:k b:k
	a * b	a + b:k a - b:k a / b:k	a % b:k a:k b:k
	a / b	a + b:k a – b:k a * b:k	a % b:k a:k b:k
	a % b	a + b:k a – b:k a * b:k	a / b:k a:k b:k
Absolute	a	abs(a):abs(a+k)	-abs (a):abs(a) 0:abs(a)
Logical	a && b	a||b:min[Tfit(a)+Ffit(b), Ffit(a)+Tfit(b)] a:Tfit(a)+Ffit(b)	b:Ffit(a)+Tfit(b) true:min [Ffit(a), Ffit(b)] false:Tfit(a)+Tfit(b)
	a || b	a&&b:min[Tfit(a)+ Ffit(b), Ffit(a)+Tfit(b)] a:Ffit(a)+Tfit(b)	b:Tfit(a)+Ffit(b) true:Ffit(a)+Ffit(b) false:min[Tfit(a), Tfit(b)]

Fitness function

- **Reach Distance**
  - *2 * approach level + normalized (branch distance)*
- **Mutation Distance**
  - *normalized (mutation distance) + normalized (pdm)*
    - *pmd = min[Tfit(O) + Ffit (M), Tfit (M) + Ffit (O)]*
    - *(Original pred == T && Mutated pred == F)* || *(Original pred == F && Mutated pred == T)*
- **Impact Distance**
  - *approach level + normalized (branch distance)*

Fitness function (*Impact Distance*)

- **Observation**
  - *Mutants are exposed when they impact some **specific program nodes**.*
  - ***Targeting some nodes** of the mutant program when having mutants weakly but not strongly killed is **likely to impact** these nodes.*
  - *Incremental search (reach, infect, propagate)*
- **Ranks the program nodes according to their ability to reveal mutants**
  - *Computes a **ratio of the killed over the live** mutants when they are impacted.*

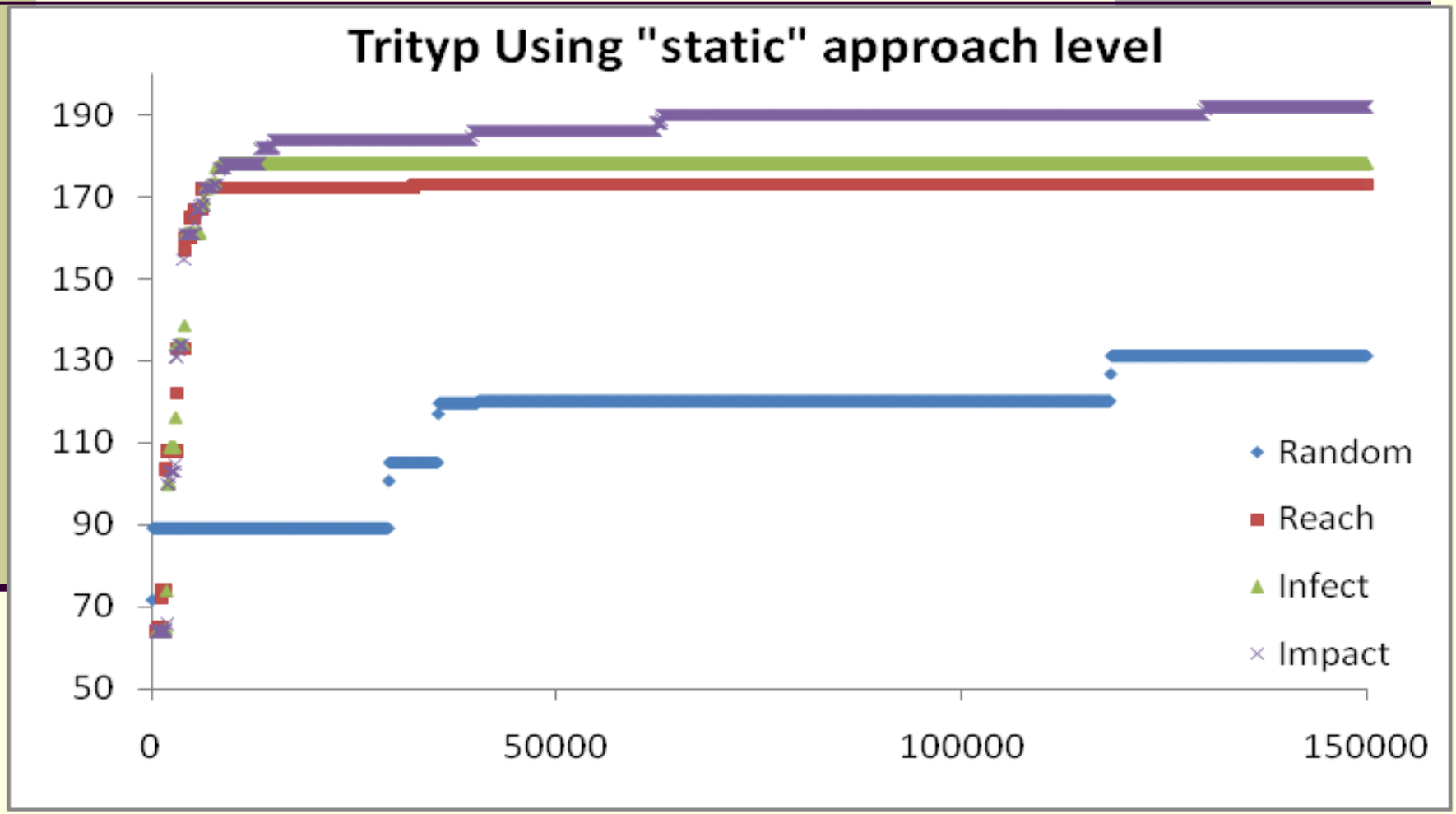
Dynamic approach level

- **Approximation of the approach level based on dynamic program execution**
  - *Intersection of all the nodes that are contained in all the encountered execution paths that reach a targeted node.*
  - *Mechanism for producing new tests based on the combined use of the encountered execution paths.*
- **Record the program execution paths encountered during the search process**
  - *Many program paths are encountered collaterally*

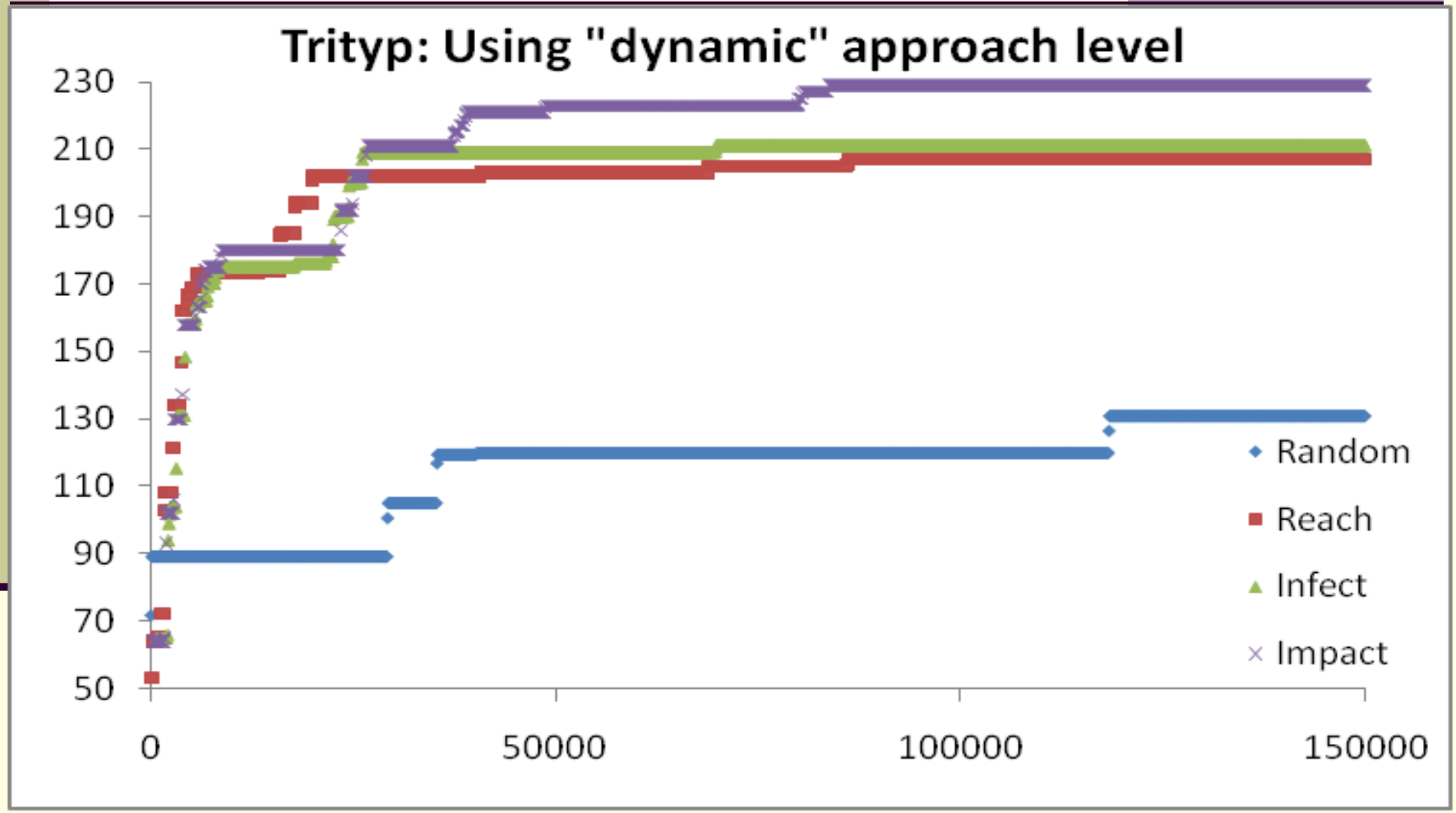
Case Study

- **Search based (*Strong mutation*)**
  - *Comparison of the Random, Reach, Infect and Impact fitness.*
  - *Comparison when using Dynamic approach level*
  - *ABS, AOR, ROR and LCR operators*
  - *Hill climbing approach (AVM)*
  - *Maximum 50,000 fitness evaluations per introduced mutant*

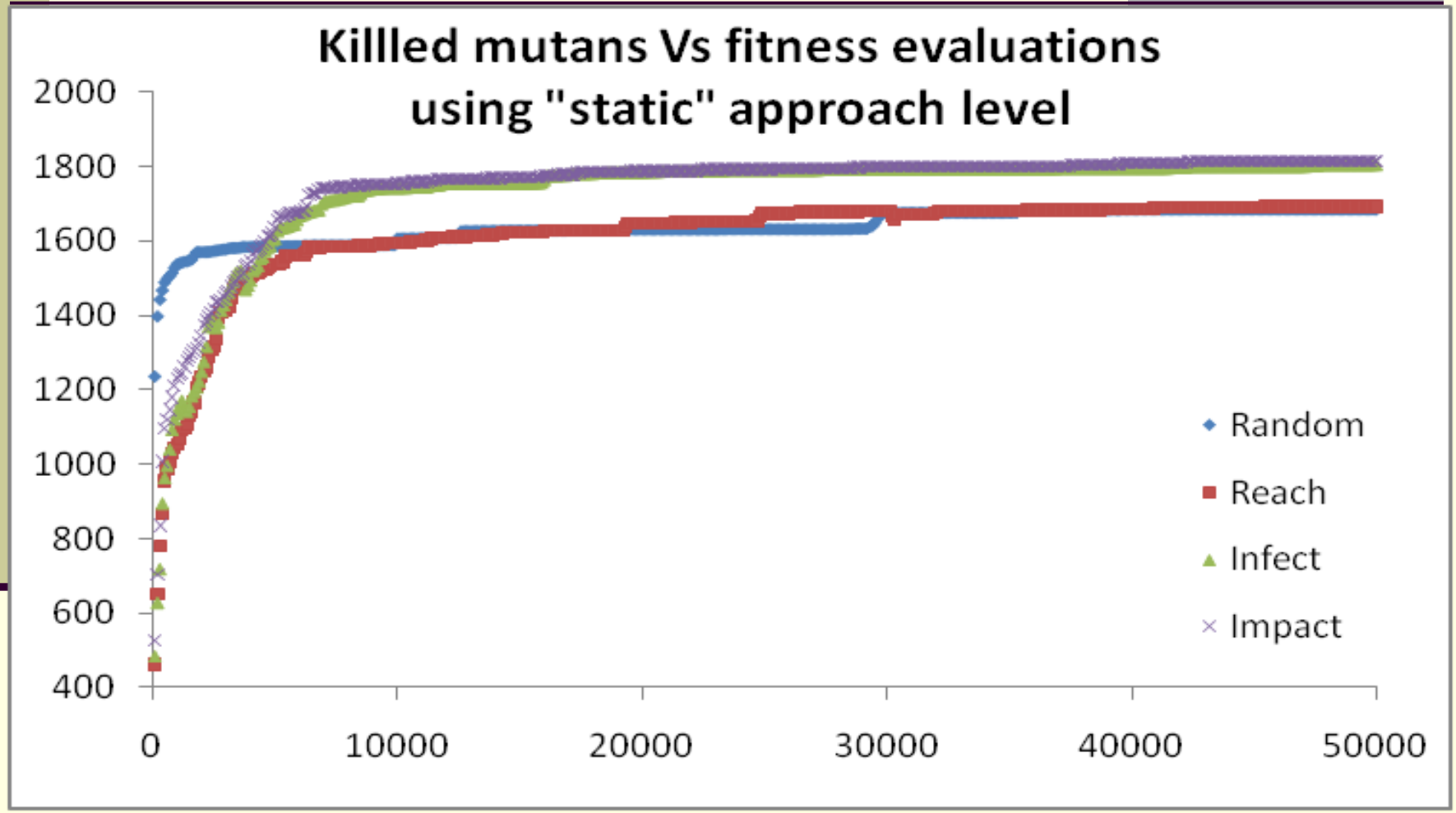
Search Based Study-*Results*



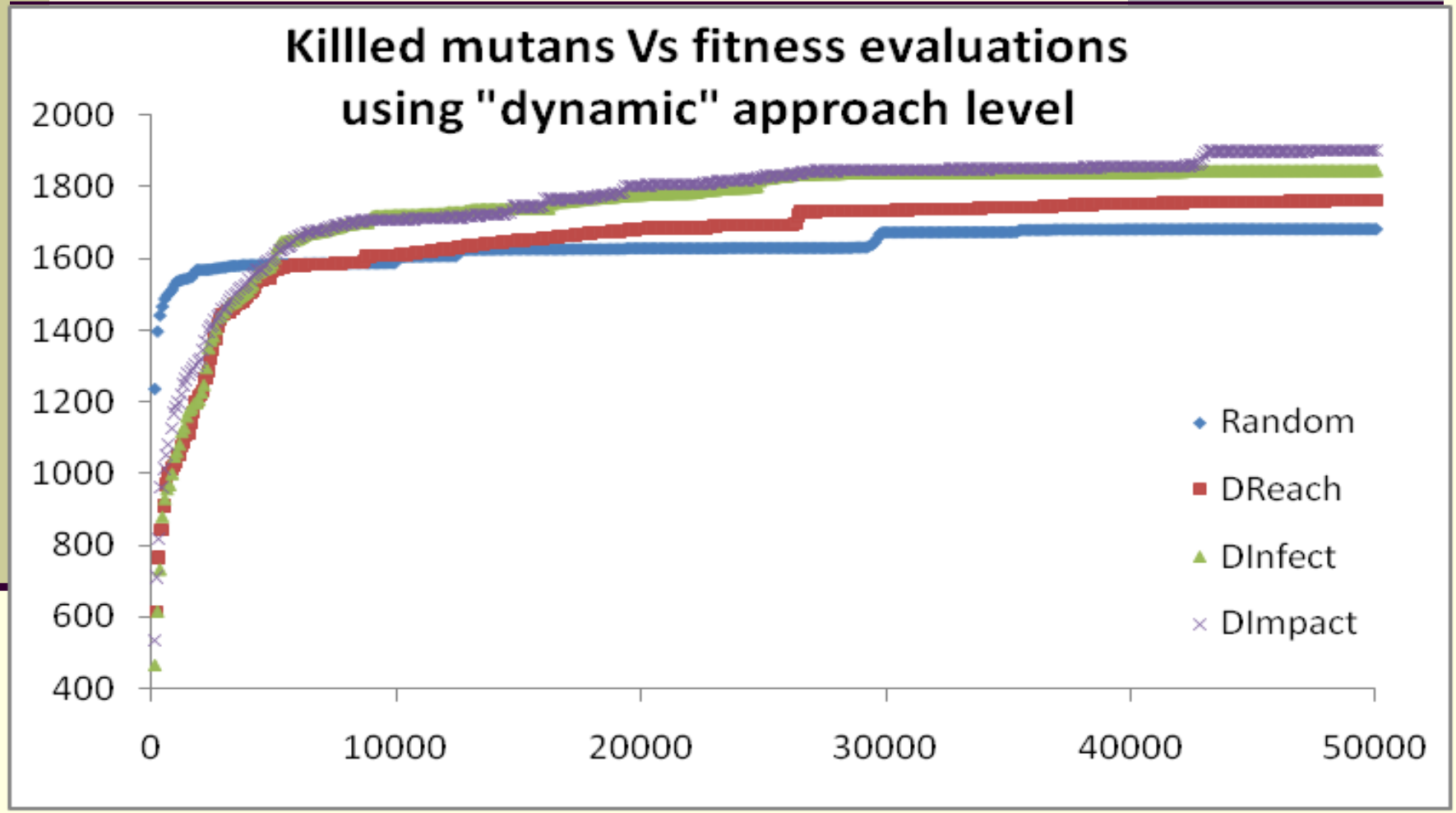
Search Based Study-*Results*



Search Based Study-*Results*



Search Based Study-*Results*



Search Based Study-*Results*

- **Fitness functions results**
  - *No. of killed mutants per fitness*

Test Subject	Random	Reach	Infect	Impact	DReach	DInfect	DImpact
Triangle	102.2	94	103	103.4	96.4	103	103.2
Tritype	125.6	173.8	178.4	184.8	205.4	210.4	223
Triangle	102	131	144.4	146.2	143.8	148.6	185
Remainder	205.8	201.4	206	206	201.4	206	206
Callendar	189	165	195.2	193.2	168.6	198.8	200
Cancel	712.6	686.2	732.2	732.6	709.26	732	733.2
FourBalls	187.2	183.2	185	186.8	181	185.8	188
Quadratic	59.07	58	61.22	61.8	58	60.6	63

Conclusion

- **Mutation based test case generation**
  - *Use of the AVM method for killing mutants*
  - *Better fitness than previous attempts*
  - *Approximation of the mutant sufficiency condition*
- **Dynamic approach level improves the effectiveness of all the utilized fitness functions**
  - *Helps overcoming difficulties of the static approach level*
  - *Helps generating test cases based on the existing ones or previously produced.*

Future Directions

- **New fitness functions**
  - *Approximate sufficient condition*
- **Equivalent mutants**
  - *Dynamic identification of (likely to be) equivalent mutants*
- **Use dynamic approach level for regression testing**
  - *Efficiently generate new tests based on the existing ones*

Thank you for your attention…

Questions ?

Contact

Mike Papadakis *mpapad@aueb.gr*

References

- *Mike Papadakis and Nicos Malevris. "**Automatic Mutation based Test Data Generation**", in Annual conference on Genetic and evolutionary computation, (GECCO'11), Dublin, Ireland, July 2011. (Poster publication)*
- *Mike Papadakis and Nicos Malevris. "**Automatic Mutation Test Case Generation Via Dynamic Symbolic Execution**", in 21st International Symposium on Software Reliability Engineering (ISSRE'10), San Jose, California, USA, November 2010.*
- *Mike Papadakis and Nicos Malevris. "**Metallaxis an Automated Framework for Weak Mutation**", Technical Report, http://pages.cs.aueb.gr/~mpapad/TR/MetallaxisTR.pdf.*
- *Mike Papadakis and Nicos Malevris. "**Automatically Performing Weak Mutation with the Aid of: Symbolic Execution, Concolic and Search Based Testing**", in Software Quality Journal. (to appear)*