
Software Test Data Generation Using Evolution

Gursaran

Reader

Dept. of Mathematics

Dayalbagh Educational Institute

Outline

- Test Adequacy Criterion
- Test Data Generation Problem
- Evolutionary Algorithms
- Evolutionary Algorithms for Test Data Generation
- Testing Object-Oriented Software
- Test Data Generation using UML and Genetic Algorithms

Terminology

- **Fault**
 - A defect in the software
- **Error**
 - Defect in the output
- **Failure**
 - An execution that produces an error
- **Test Case**
 - An input on which a program is executed during testing
- **Test Set**
 - A set of test cases for testing a program

Test Data Adequacy Criterion

- Testing can only detect the presence of faults but not their absence.
- Central questions in software testing
 - What is the test criterion?
 - What constitutes an adequate test?

Test Data Adequacy Criteria

- A predicate
- Specifies testing requirements
- Requirements on Test Data Adequacy Criteria
 - Reliability
 - Validity
- Not practically applicable

Test Data Adequacy Criteria as Stopping Rules

■ Let

- P be a set of programs
- S be a set of specifications
- D be the set of inputs of the programs in P
- T be the class of test sets

A test data adequacy criterion C is a function

$$C: P \times S \times T \rightarrow \{true, false\}$$

$C(p, s, t) = true$ means that t is adequate for testing program p against specification s according to the criterion C , otherwise t is inadequate.

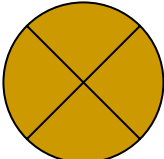
Test Data Adequacy Criteria as Measurements

A test data adequacy criterion is a function C ,

$$C: P \times S \times T \rightarrow [0,1]$$

$C(p, s, t) = r$ means that the adequacy of testing the program p by the test set t with respect to the specification s is of degree r according to the criterion C . The greater the real number r , the more adequate the testing.

Classification

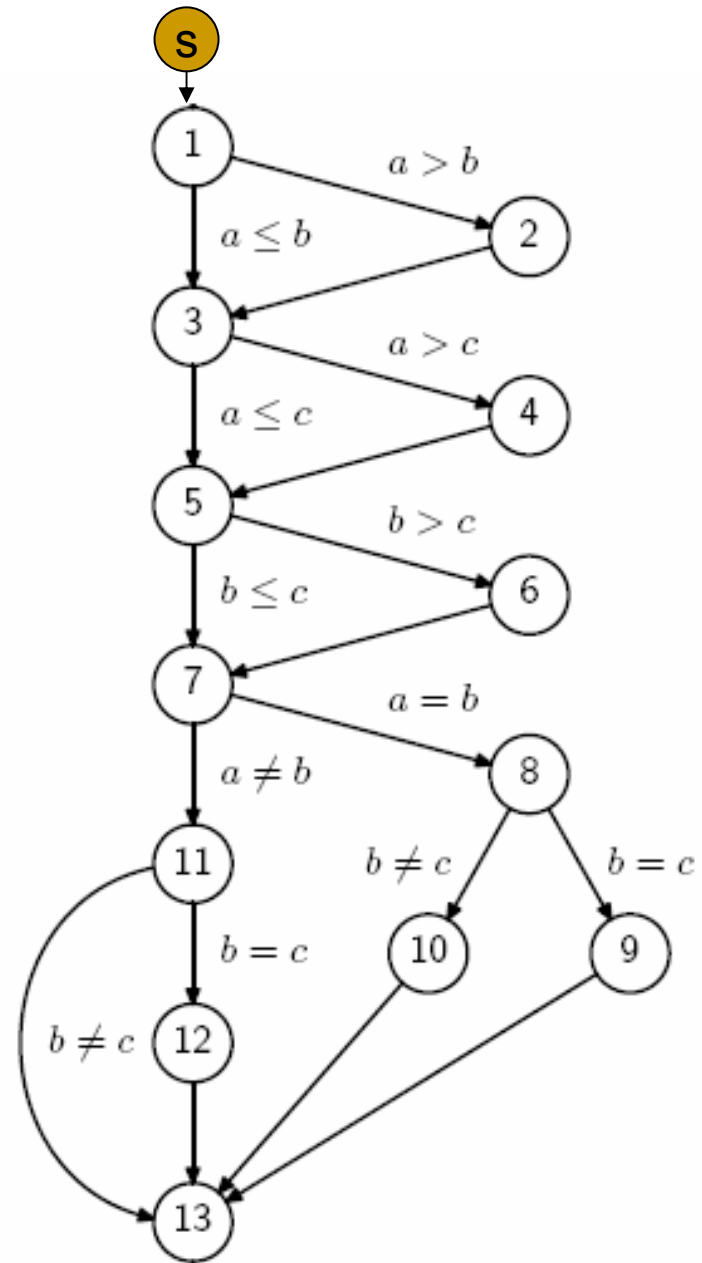
Program Based			
Specification Based			
	Structural	Fault Based	Error Based

Control Flow Graph

- Represents program structure
- A CFG is a directed graph $C = (N, E, s, e)$ where
 - N is a set of nodes
 - $E = \{(n, m) \mid n, m \in N\}$ connecting the nodes
 - s is unique entry node and e is unique exit node
- Each node is defined as a *basic block*:
 - An uninterrupted consecutive sequence of instructions, where the flow of control enters in the beginning and leaves at the end without halt or possibility of branching except at the end.
- Edge (n, m) represents possible transfer of control from n to m
- An edge (n, m) is called a *branch* if n is a test node
- A branch can be labeled by a *branch predicate* describing the conditions under which the branch would be traversed

CFG Node

```
s   int triType(int a, int b, int c) {  
1   int type = PLAIN;  
1   if (a > b)  
2       swap(a, b);  
3   if (a > c)  
4       swap(a, c);  
5   if (b > c)  
6       swap(b, c);  
7   if (a == b) {  
8       if (b == c)  
9           type = EQUILATERAL;  
   else  
10      type = ISOSCELES;  
   }  
11  else if (b == c)  
12      type = ISOSCELES;  
13/e return type;  
}
```



Definitions

- An **input vector** I is a vector $I = (x_1, x_2, \dots, x_k)$ of input variables to a program.
- The **domain of an input variable** x_i , $1 \leq i \leq k$, is the set of all values that x_i can take on.
- The **domain of a program** is the cross product $D = D_{x_1} \times D_{x_2} \times \dots \times D_{x_k}$ where each D_{x_i} is the domain of the input variable x_i .
- A **program input** x is a single point in the k -dimensional input space D , $x \in D$.
- A **path** through a CFG is a sequence $P = \langle n_1, n_2, \dots, n_m \rangle$ such that for i , $1 \leq i < m$, $(n_i, n_{i+1}) \in E$.
- A path is **feasible** if there exists a program input x for which the path is traversed during program execution.

Program Based Structural Test Adequacy Criteria

Program Based	Node Coverage Branch Coverage Path Coverage		
Specification Based			
	Structural	Fault Based	Error Based

Applicability

- **Finite Applicability** – Requirement that an adequacy criterion can always be satisfied by a finite test set.
 - Problems
 - Infinite number of different paths
 - Infeasibility
- Limit testing to feasible versions of adequacy criteria
- Leads to the *undecidability* problem
 - Whether a statement in a program is feasible or not is undecidable
 - It is not possible to take an adequacy criterion and determine whether an input exists that satisfies it.

Automated Test Data Generation: A Compromise

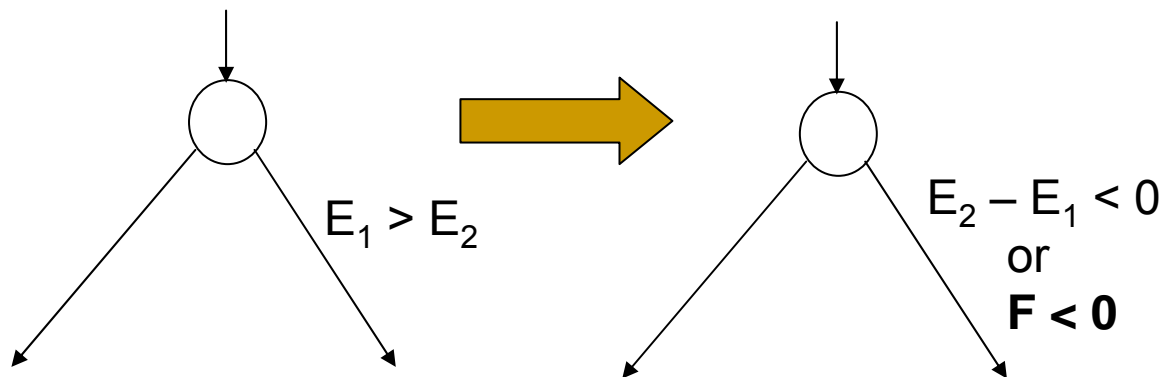
- The *undecidable* problem
- Attempt under resource limitations
 - If resources are exhausted, the attempt fails
- What is the best that we can do within the allocated resources?

Program Based Test Data Generation Problem

- Let $P = \langle n_1, n_2, \dots, n_m \rangle$ be a path in the program. The goal of test data generation problem is to find a program input $\mathbf{x} \in D$ on which P is traversed

Problem Restatement

- Reduce the problem of test data generation to a sequence of subgoals each of which can be solved using function minimisation
- Assume *simple* relational expressions involving arithmetic expressions



Problem Restatement

- Branch predicates can be transformed into an equivalent predicate

$F(x) \text{ rel } 0, \text{ rel} \in \{ <, \leq, = \}, x \in D, \text{ where}$

$$F(x) = \begin{cases} \text{positive (or 0 with } <), & \text{when Branch Predicate is false} \\ \text{negative (or 0 with } = \text{ or } \leq), & \text{when Branch Predicate is true} \end{cases}$$

Problem Restatement

- The exact nature of $F(x)$ need not be known

Program \mathcal{P} fragment:

if ($p \geq 21$) ...

Goal:

To ensure that the TRUE branch is taken

Execute \mathcal{P} on input x :

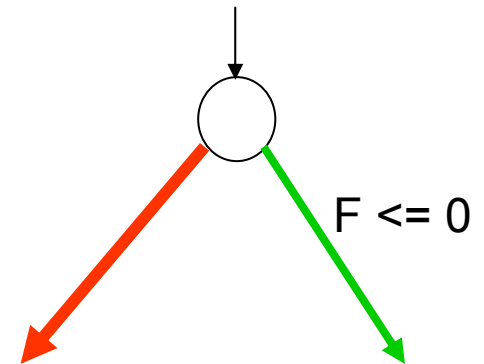
Let p_v is the value that p takes

Problem Restatement

- The function F is

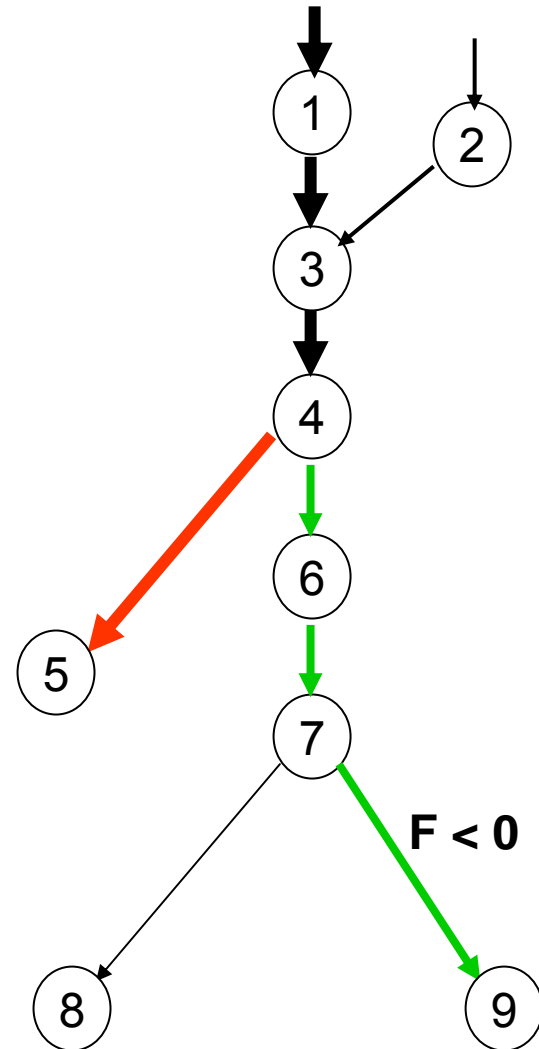
$$F(x) = \begin{cases} 21 - p_v, & \text{when FALSE branch is taken} \\ 0 \text{ or negative,} & \text{when TRUE branch is taken} \end{cases}$$

- Problem of test data generation is reduced to function minimisation:
 - Find an input x that minimizes $F(x)$



Problem Restatement

- Constraints?
- The Branch Condition may not be reached for certain inputs
 - Treat the problem of reaching the branch condition as a subgoal before minimizing $F(x)$



Dynamic Test Data Generation

- The problem of test data generation is reduced to that of function minimisation or maximisation.
- The source code is instrumented to collect information about the program as it executes.
- Collected information is used to measure how close the test data is to satisfying the test requirements.
- The measure is then used to modify the input to progressively move towards satisfying the test requirement.

We can use Evolutionary Algorithms here

Evolutionary Algorithms

- Evolutionary algorithms are stochastic search methods that model natural biological evolution.
- Operate on a “population” of candidate solutions applying the principle of survival of the fittest to produce better and better approximations to a solution.
- At each generation, solutions with high "fitness" are "recombined" with other solutions by swapping parts of a solution with another. Solutions are also "mutated" by making a small change to a single element of the solution .
- This process leads to the evolution of populations of individuals that are better adapted to their environment.

Evolutionary Algorithms

```
procedure EA {  
    t = 0;  
    initialize population P(t);  
    evaluate P(t);  
    until (done) {  
        t = t + 1;  
        parent_selection P(t);  
        recombine P(t);  
        mutate P(t);  
        evaluate P(t);  
        survive P(t);  
    }  
}
```

Evolutionary Algorithms

- An evolutionary algorithm typically **initializes** its population randomly,
 - Domain specific knowledge can also be used to bias the search.
- **Evaluation** measures the fitness of each individual according to its worth in some environment.
- Selection
 - **Parent selection** - Decide who becomes parents and how many children do they have
 - **Survival** - Decide who survives in the population
- Children are created via **recombination**, which exchanges information between parents.
- **Mutation** perturbs the children.

Types of Evolutionary Algorithms

- Evolutionary Programming
- Evolution Strategies
- Genetic Algorithms
- Genetic Programming

Genetic Algorithms

- Traditionally use a bit-string representation – more domain independent
- Parents are selected according to a probabilistic function based on relative fitness
- N children are created by recombining N parents
- N children are mutated and survive replacing the parents
- Emphasis on mutation and crossover is opposite to other variants of EA
- Recombination is the primary search operator

Application of GAs to Software Test Data Generation

- Representation of sample tests
 - Assuming scalar input variables x_i
 - Each variable x_i is represented as a bit string
 - Input vector $I = (x_1, x_2, \dots, x_k)$ is represented as concatenation of bit string representations of x_1, x_2, \dots, x_k in order
- Population is then a set of bit strings representing test cases
 - Size of the population is proportional to the number of bits in the representation of I .
- Fitness function
 - Formulation based on required coverage

Branch Coverage

- Each individual branch is taken as the target of a test data search

Branch Coverage: Fitness Function

- Assuming that TRUE branch is to be covered

Decision Type	Example	Fitness Function
inequality	if (c >= d)	$F(x) = \begin{cases} d - c, & \text{if } d > c \\ 0, & \text{otherwise} \end{cases}$
equality	if (c == d)	$F(x) = d - c $
true/ false value	if (c)	$F(x) = \begin{cases} 1000, & \text{if } c \text{ is FALSE} \\ 0, & \text{otherwise} \end{cases}$

Example

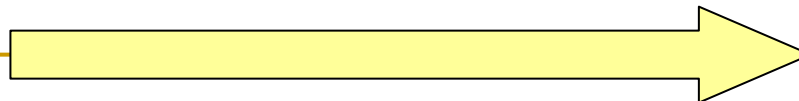
CFG Node

```
(s)    main(int argc, char **argv)
      {
(1)        int a = atoi(argv[1]);
(2)        if (a > 100)
(3)            printf("Hello World");    // Target
(e)    }
```

Input	100 – a	Fitness	Normalised Fitness
94	6	1 / 6	0.552
91	9	1 / 9	0.368
49	51	1 / 51	0.065
-112	212	1 / 212	0.015

91 : 01011011 010 | 11011 010 | 00000 01000000 : 64
 94 : 01100000 011 | 00000 011 | 11011 01111011 : 123

 91 : 01011011 01011 | 011 01011 | 000 01011000 : 88
 -112 : 10010000 10010 | 000 10010 | 011 10010011 : -109



Reaching the Target Condition

- In dynamic test generation, function minimization cannot be performed unless the flow of control reaches a certain point in the code.
- Solution:
 - Delay attempts to satisfy a certain condition until tests have been found tests that reach that condition
 - A table is generated to keep track of the conditional branches already covered by existing test cases.
 - **If neither branch of a condition has been taken** - function minimization cannot be applied to that condition.
 - **If both branches have been taken** - coverage is satisfied for that condition.
 - **If only one branch of a condition has been exercised** - condition has been reached - apply function minimization in search of an input that will exercise the other branch.

Reaching the Target Condition

■ Critical Branch

- A critical branch is simply a branch which leads to the structural target of interest being missed in a path through the program

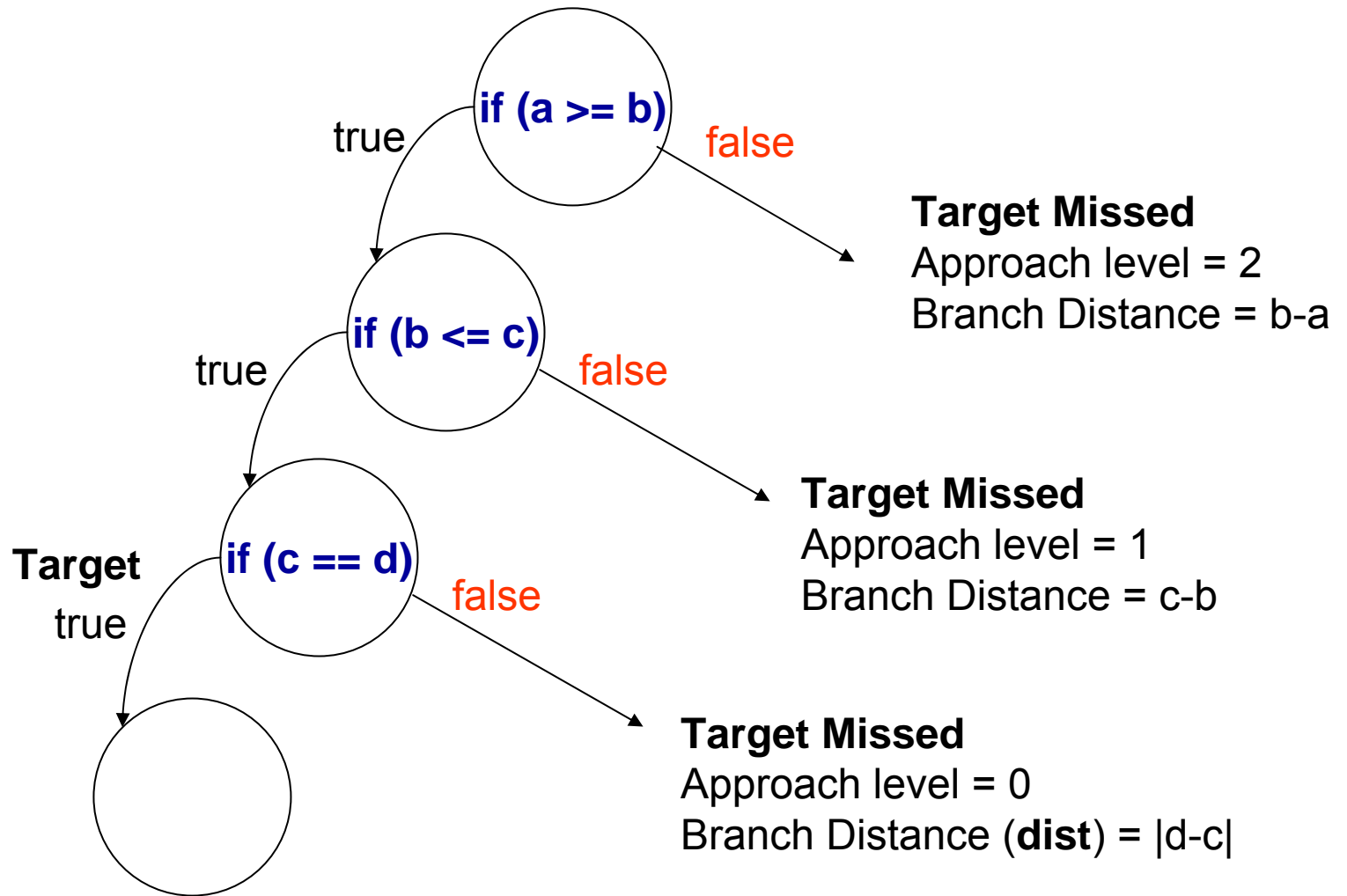
■ Fitness function is made of two components

□ Approach level

- Subtract one from the number of critical branches lying between the node from which the individual diverged away from the target, and the target itself.

□ Branch distance

- At the point where control flow took a critical branch for some individual, the *branch distance* is calculated



$$\text{normalize}(\text{dist}) = 1 - 1.001^{-\text{dist}}$$

$$\text{fitness}(I) = \text{approach level} + \text{normalize}(\text{dist})$$

Problems : Deceptive Functions

CFG Node

(s) double function_under_test(double x)

{

(1) if (inverse(x) == 0)

 {

(2) **// target**

 }

(e) }

double inverse(double d)

{

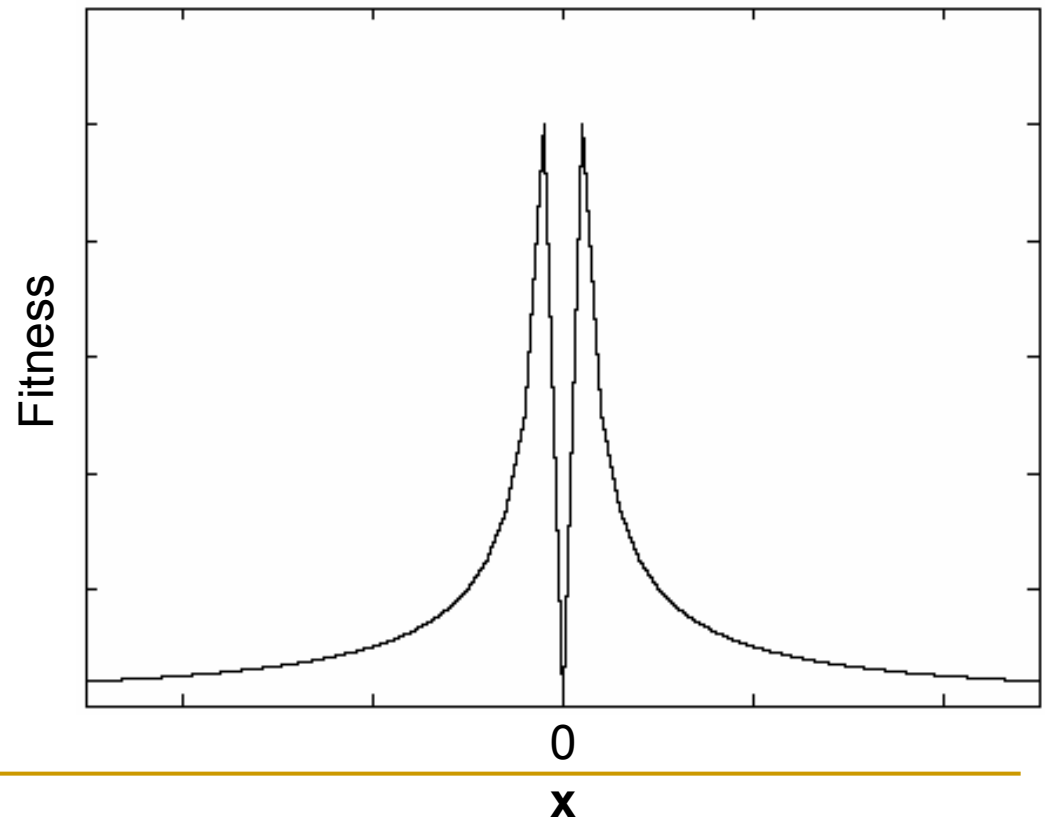
(3) if (d == 0)

(4) return 0;

 else

(5) return 1 / d;

}



Problems: Flag Variables

CFG Node

```
(s) void flag_example(int a, int b)
    {
(1)     int flag = 0;
(2)     if (a == 0)
(3)         flag = 1;
(4)     if (b != 0)
(5)         flag = 0;
(6)     if (flag)
        {
(7)         // target
        }
(e) }
```

Object-Oriented Software Testing: Problems

■ The Class and Object

- ❑ A class can only be tested indirectly by testing its instances
- ❑ There is no sequential flow of control within an object; control flow is characterised by the sequence of messages that the object receives
- ❑ Object under test is by definition not the object under operation

■ Encapsulation

- ❑ Problem of *observability*
- ❑ Abstract classes and templates cannot be tested directly

Object-Oriented Software Testing: Problems

■ Inheritance

- ❑ A class cannot be considered as tested even if the classes from which it inherits have been tested
- ❑ Name resolution in multiple inheritance

■ Polymorphism

- ❑ Introduces undecidability and it is not possible to predict until run time what code will be executed

Class Testing: Observations

- The state of an instance determines its behaviour on the receipt of a message.
- Given a class C , let S_C denote the state space of C , and let Ψ_C denote the **operation space**, which is the set of all possible sequences of operations allowable for C
- **Observation 1:** For any given pair of states s_1 and s_2 in S_C , there may be more than one operation sequence Ψ_C that causes a transition from s_1 to s_2 . Each of these may have a different fault exposing ability.

Class Testing: Observations

- **Observation 2:** A sequence of operations $\psi \in \Psi_C$ defines a function from S_C to S_C . Therefore, for different initial states, a given sequence of operations may exhibit different fault exposing abilities.
- **Observation 3 :** In class testing, a test suite should consist of
 - a set of properly selected sequences of operations; and
 - a set of properly selected class states.

Class Testing

- For each method
 - An object of the Class under test is created using one of the available constructors.
 - A sequence of zero or more methods is invoked on such an object to bring it to a proper state.
 - This may involve creating more objects (parameters)
 - The method currently under test is invoked.
 - The final state reached by the object being tested is examined to assess the result of the test case.

Evolutionary Testing

```
testCaseGeneration(classUnderTest: Class)
1      targetsToCover = targets(classUnderTest)
2      curPopulation = generateRandomPopulation(popSize)
3      while targetsToCover !=  $\phi$  and
           executionTime() < maxExecutionTime
4          t = selectTarget(targetsToCover), attempts = 0
5          while not covered(t) and attempts < maxAttempts
6              execute test cases in curPopulation
7              update targetsToCover
8              if covered(t) break
9              compute fitness[t] for test cases in curPopulation
10             extract newPopulation from curPopulation
                  according to fitness[t]
11             mutate newPopulation
12             curPopulation = newPopulation
13             attempts = attempts + 1
14         end while
15     end while
```

Evolutionary Testing

- Test Adequacy Criterion
 - For each method - Branch Coverage
- Fitness Function
 - Given the transitive set of all control and call dependences that lead to the given target, the proportion of such edges that is exercised during the execution of a test case measures its fitness

Evolutionary Testing

■ Test case:

- a test case is a sequence of constructor and method invocations, including parameter values.
- Assertions on the expected state after their execution completes the test case

`$a=A():$b=B():$a.m(int, $b) @ 3`

Mutation & Recombination Operators

■ Mutation

- ❑ Mutation of an input value
- ❑ Constructor change
- ❑ Insertion of method invocation
- ❑ Removal of method invocation

■ Recombination (One-point crossover)

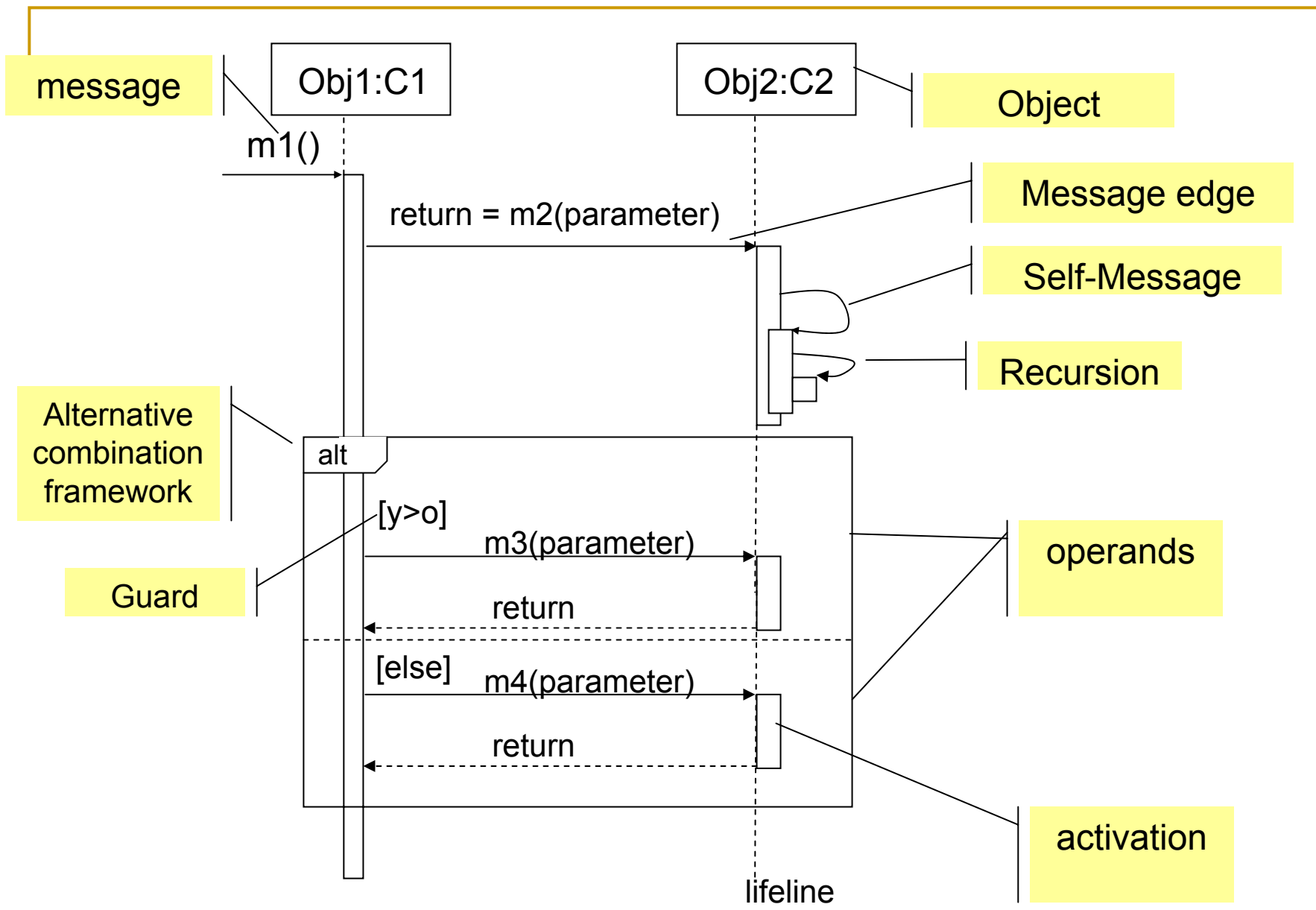
`$a=A():$b=B():$b.f():$a.m(int,$b) @1,5`

`$a=A(int,int):$b=B():$b.g():$a.m(int,$b) @0,3,4`

`$a=A():$b=B():$b.f():$b.g():$a.m(int,$b)@1,4`

`$a=A(int,int):$b=B():$a.m(int,$b) @0,3,5`

Towards Test Data Generation for Cluster Testing using UML Sequence Diagrams and Genetic Algorithms



Assumptions

- The system being tested does not embody any concurrency semantics.
- Interactions are modelled using sequence diagrams in the context of an operation or a use case at the level of a software implementation.
- The flow of control is nested, and message actions are either call, return, create or of the destroy type.
- Message parameters are scalars.
- It may be noted that these assumptions are not meant to be restrictive, but outline the scope of testing described here.

What is a test case?

- Within the context of an operation
 - The collaboration modelled by a sequence diagram is initiated with a message dispatch that results in the operation being invoked.
 - A sequence of one or more such message dispatches, each of which initiates the collaboration, is taken to define a test case.
- Note: It is important to prepend a sequence of message despatches that will bring the object to a desired state first

What is a test case?

- Within the context of a use case
 - The events that flow from the actors to the use case are the inputs that occur in a sequence as specified in the sequence diagram.
 - This sequence of inputs defines a test case.

Test Adequacy Criteria

- Since a sequence diagram specifies interactions among a collaborating cluster of objects to implement some behaviour, the implementation of the interactions that are modelled by these diagrams must be verified to assure that the correct behaviour ensues.

Message edge coverage criterion

- *Definition:* Let S be a sequence diagram and let P be the program that implements S . A test set T is said to satisfy the message edge coverage criterion if every message edge in S is traversed at least once by executing P on T .
- *Basis:* In the most basic form, we would require that each message shown in a sequence diagram be sent at least once.
- *Assumption:* program P , which could be a part of a much larger implementation, can be identified and tested independently.

Message edge and loop coverage criterion

- *Definition:* Let S be a sequence diagram and let P be the program that implements S. A test set T is said to satisfy the message edge and loop coverage criterion if every message edge in S is traversed at least once by executing P on T and all loop message edges are traversed according to the following scheme: bypassed, traversed once, traversed twice, and traversed a typical number of times.
- *Basis:* Loops must be tested adequately.

All end-to-end scenario paths criterion

- *An **end-to-end scenario path** is defined to be a sequence of edges that begins with an edge that represents a message dispatch that initiates the interaction, ends with an edge that represents the last message dispatched in the interaction, and that represents a sequence of messages dispatched in one scenario of execution.*
- *Definition: Let S be a sequence diagram and let P be the program that implements S. A test set T is said to satisfy the all end-to-end scenario path coverage criterion if every end-to-end scenario path in S is traversed at least once by executing P on T.*
- *Basis: All scenarios -- specific sequence of actions that illustrates behaviour -- that make up a use case must be tested.*

The Basic Procedure

(Let P be a program that implements a sequence diagram S)

- Choose an appropriate test adequacy criterion on S .
- Setup the GA.
 - Select a chromosome representation for test case to be input to program P .
 - Select a fitness function.
 - Instrument the program P using S to create program P_t . (Since S is assumed to model interactions in P , the instrumented program P_t is used directly for test data generation).
 - Select suitable GA parameters.
- Generate test data.
 - Run the GA for test data generation using P_t for fitness function computation.
 - Identify and eliminate infeasibility.
 - Regenerate test data if necessary.

Chromosome Representation

- A chromosome should represent a test case.
- If context is an operation
 - The arguments for each call to the member function implementation corresponding to the operation under test must be represented.

Fitness Function – I (Fit – I)

(Message Edge Coverage Criterion)

- The fitness function is based on the guards of the operands in an alternative combination framework.

Fitness Function – II (Fit-II)

- if (*chromosome (test case) results in a new message edge being traversed*)

Fitness = Number of distinct, new, message edges traversed

+

the distinct message edges traversed over all preceding generations;

else

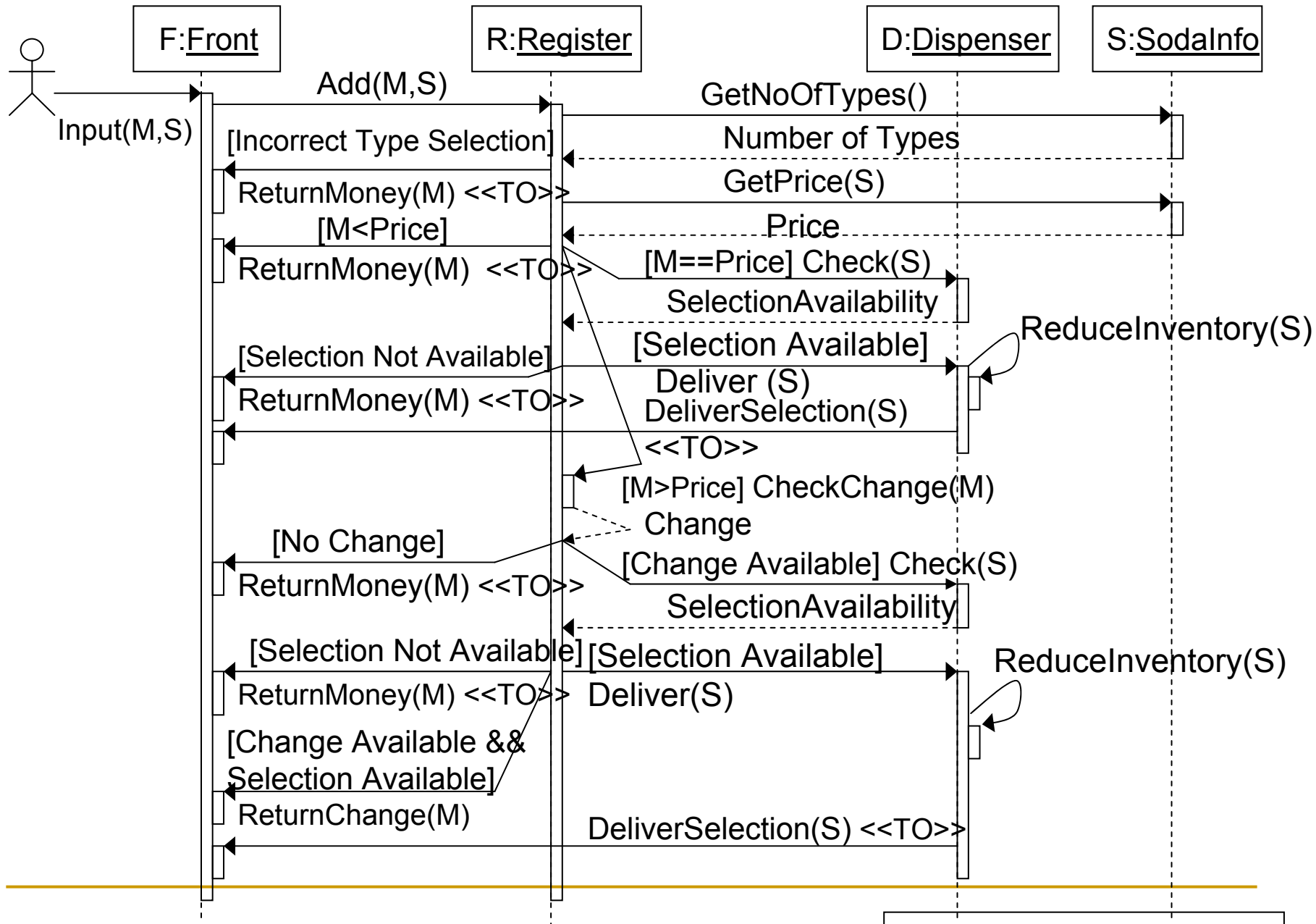
Fitness = Count of the distinct message edges traversed by the test case.

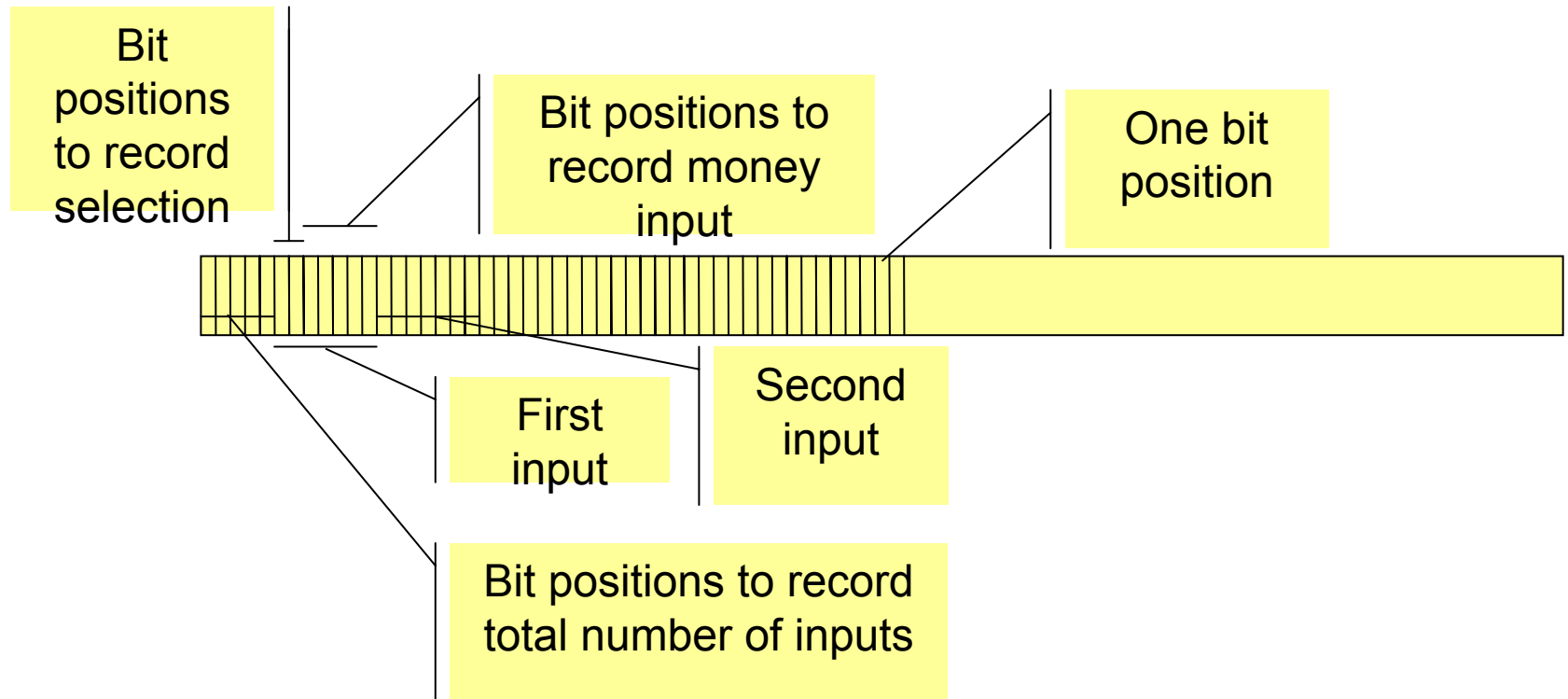
Dealing with Infeasibility

- Two schemes are possible
- If the GA is attempting to traverse a message edge, but is unable to do so over a sufficiently large, predetermined, number of generations then
 - The GA run is aborted and the message edge is manually examined for infeasibility.
 - How this may be done is not discussed here.
 - If the edge is found to be infeasible then it is marked as traversed and the GA is rerun.
- Alternatively
 - Message edge is recorded in a separate file and marked traversed.
 - After a traversal of all the message edges has been attempted, the test data generation process outputs the test set together with the percentage of test requirements that are satisfied by the test set.

Case Study

Soda Vending Machine

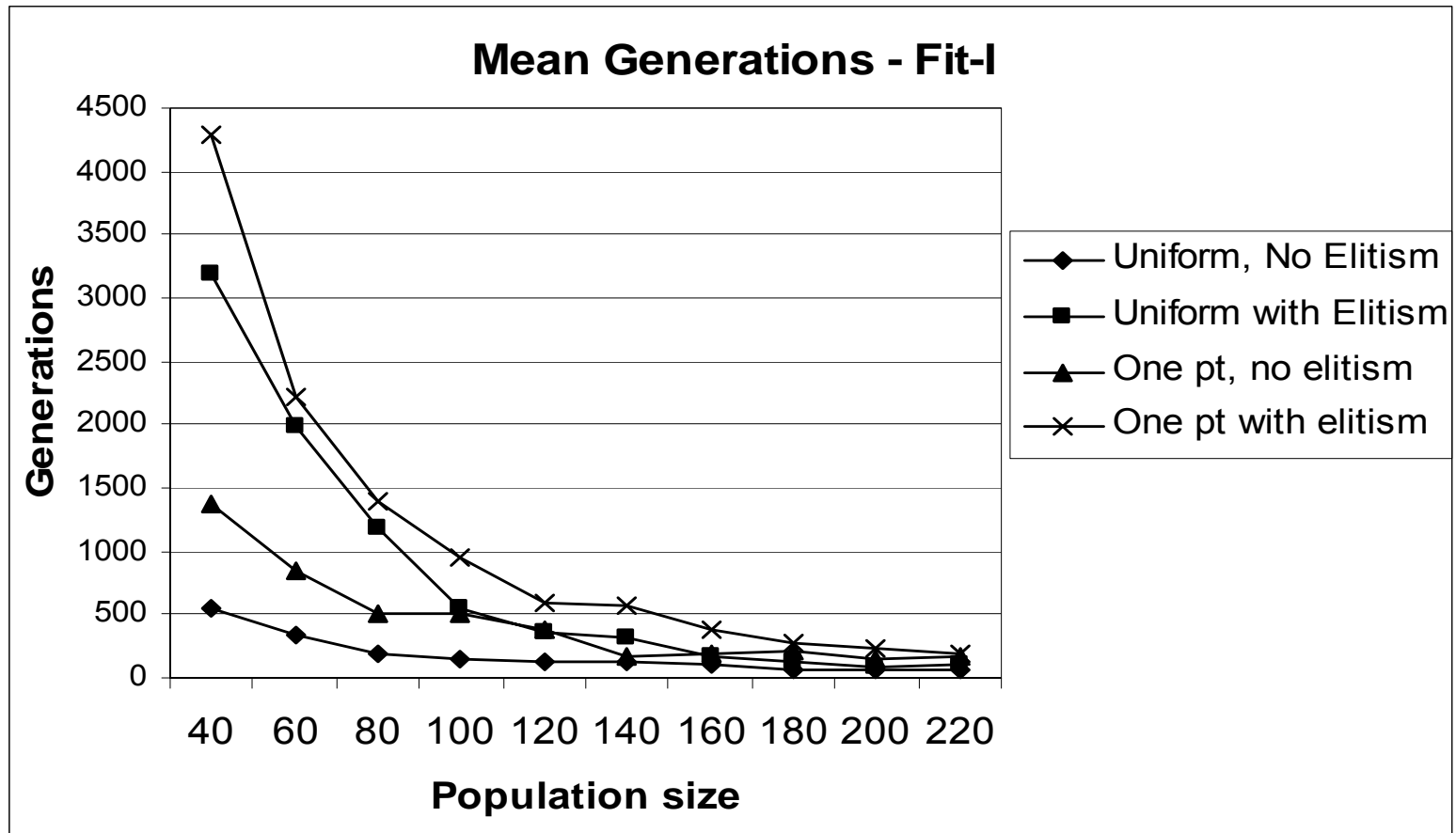




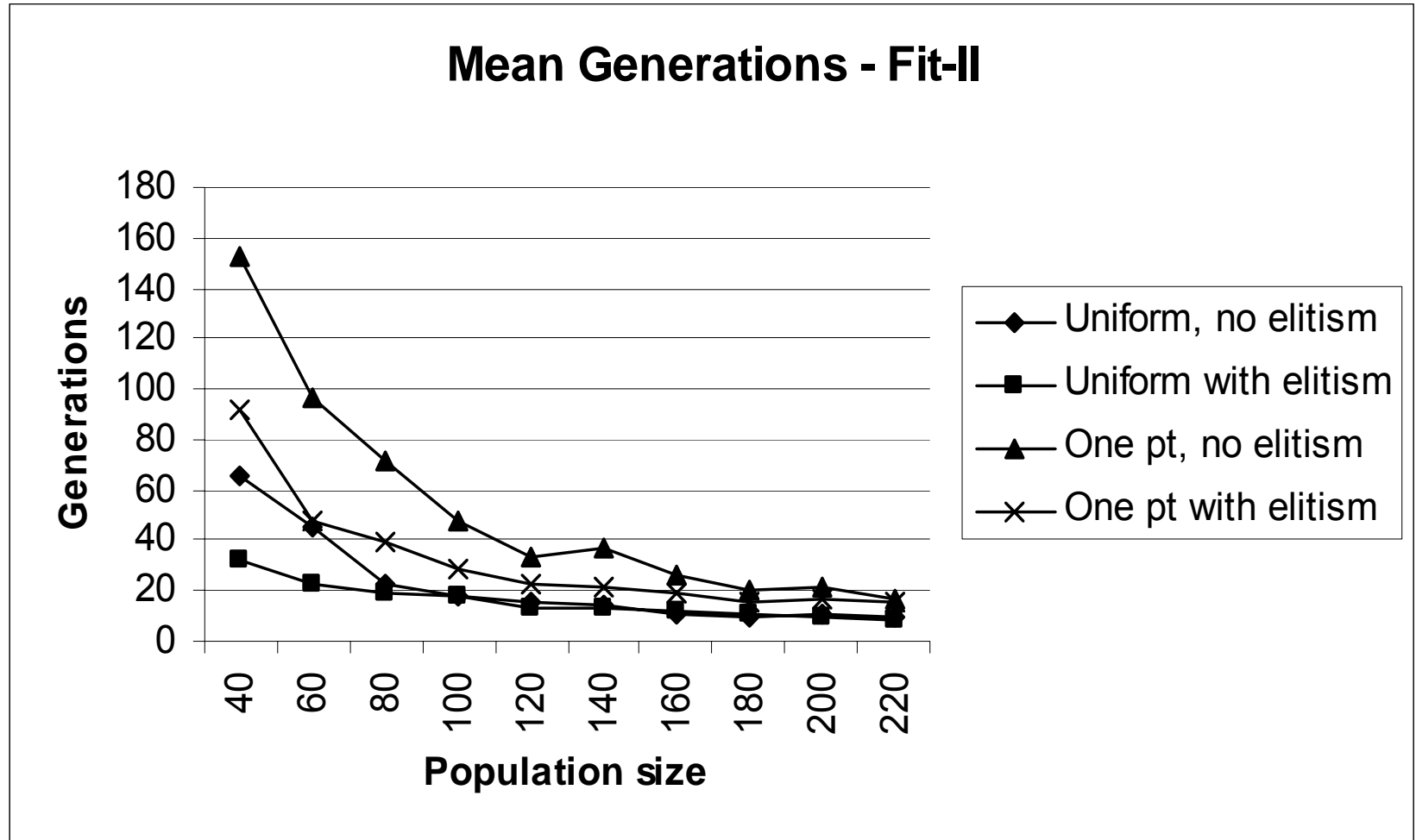
GA Parameters

<i>Machine Parameters</i>			
Number of types of soda	2; labelled 0, and 1	Initial quantities of each soda type	10
Cost of each soda type	Type 0 – Rs.10; Type 1 – Rs. 15.	Change Reserve Denominations	Rs.1; Rs.2; Rs.5; Rs.10; Rs.20
<i>GA Parameters</i>			
Chromosome Structure (number of bits)	No of Inputs – 5 Selection – 2 Amount – 5 Length -230	Selection Scheme: Default:	With elitism, and without elitism Roulette Wheel
Crossover Probability	1.0	Mutation Probability	0.005
Crossover Type	One point, uniform	Number of Experiments	100
Population size	40, 60, 80, 100, 120, 140, 160, 180, 200, and 220		

Mean Generations: Fit-I



Mean Generations: Fit - II



References

- Edvardsson, J., “A Survey on Automatic Test Data Generation,” in *Proceedings of the Second Conference on Computer Science and Engineering*, ECSEL, Linköping, pp. 21-29, October 1999.
- Gu, D., Y. Zhong and S. Ali, On Testing of Classes in Object-Oriented Programs, www.cs.ubc.ca/local/reading/proceedings/cascon94/papers/gu.ps
- Gursaran, Rabins Porwal, and Rahul Caprihan, “Towards Test Data Generation for Cluster Testing using UML Sequence Diagrams and Genetic Algorithms”, in *Proceedings of the Sixth International Conference on Information Technology, CIT-2003*, Bhubaneswar, India, Dec 22-25, pp.61-66, 2003.
- Korel, B., “Automated Software Test Generation”, *IEEE Trans. on Software Engineering*, vol. 16(8), pp.870-879, 1990.
- McMinn, P., M. Holcombe, “Evolutionary Testing Using an Extended Chaining Approach,” *Evolutionary Computation*, 14(1), pp. 41-64, 2006.
- Michael, C.C., G.E.McGraw, and M.A.Schatz, “Generating Test Data by Evolution”, *IEEE Trans. on Software Engg.*, vol. 27(12), pp.1085-1110, 2001.
- Porwal R., *Testing Classes in Object-Oriented Software: Issues, Analysis and Approaches*, Ph.D. Thesis, Dayalbagh Educational Institute, Agra, 2004.
- Spears, W.M., K A. De Jong, T. Bäck, D.B. Fogel, H. de Garis, Spears, “An Overview of Evolutionary Computation,” in *Proceedings of the European Conference on Machine Learning*, v667, pp.442-459, 1993.
- Tonella, P., “Evolutionary Testing of Classes,” *ISSTA'04*, Boston, Massachusetts, USA, July 11–14, pp.119-128, 2004.
- Wall, M., *GAlib: A C++ Library of Genetic Algorithm Components*, version 2.4, <http://lancet.mit.edu/ga/>, 1996.
- Zhu, H., P.A.V. Hall, and J.H. May, “Software Unit Test Coverage and Adequacy”, *ACM Computing Surveys*, vol. 29(4), pp.366-427, 1997.

Thank You

Questions?