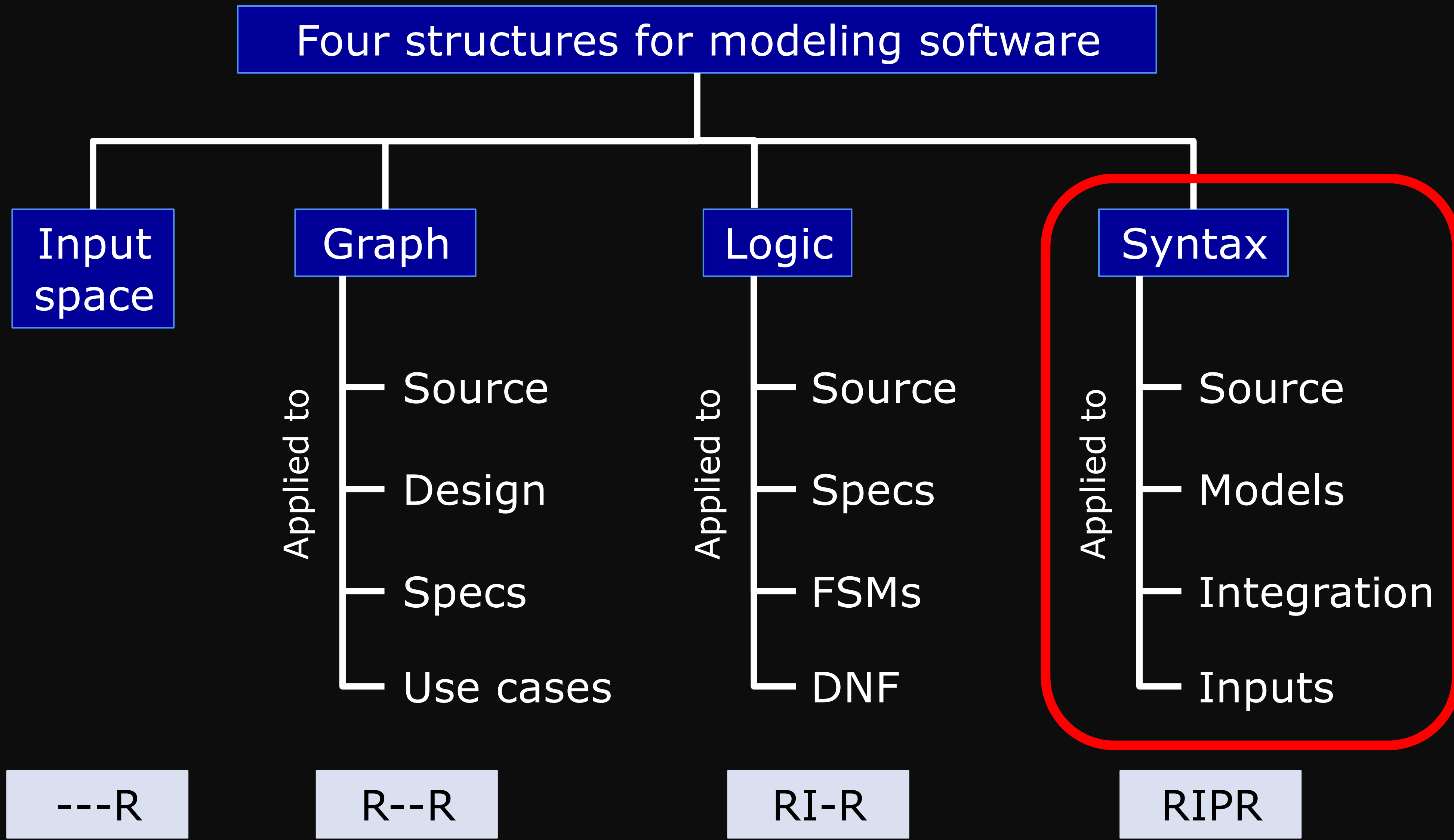
Syntax-based Testing

CS 3250
Software Testing

[Ammann and Offutt, "Introduction to Software Testing," Ch. 9.1]

Structures for Criteria-Based Testing



ISP, Graph, Logic, and Syntax

```
# Return index of the first occurrence of a letter in string,
# Otherwise, return -1   (note: faulty version)

def get_index_of(string, letter):
    index = -1
    for i in range(1, len(string)):
        if string[i] == letter:
            return i
    return index
```

Software artifact

Syntax (Grammar-based Testing)

for_stmt : 'for' exprlist 'in' testlist ':' suite ['else' ':' suite]
exprlist : (expr|star_expr) (',' (expr|star_expr))* [',']
testlist : test (',' test)* [',']
suite : simple_stmt | …
…

Syntax (Program-based Mutation)

```
def get_index_of(string, letter):
    index = -1
    for i in range(1, len(string)):
△    if string[i] != letter:
            return i
    return index
```

Syntax-Based Testing

- Rely on syntactic description of software artifacts

- Syntactic descriptions can come from many sources:
  - Programs
  - Integration elements
  - Design documents
  - Input descriptions

- Tests are created with two general goals
  - Cover the syntax in some way
    - Generate artifacts that are valid (correct syntax)
  - Violate the syntax
    - Generate artifacts that are invalid (incorrect syntax)

Grammar-Based Coverage Criteria

- Common practice: uses automata theory to describe software artifacts
  - BNF – describe programming languages
  - Finite state machines – describe program behavior
  - Grammars and regular expressions – describe allowable inputs
- Focus:
  - Testing the program with valid inputs
    - Exercise productions of the grammar according to some criterion
  - Testing the program with invalid inputs
    - Use grammar-based mutation to test the program with invalid input

Grammar: Regular Expression

(G s n | B t n)*

Closure operator

zero or more occurrences

Choice

Either one can be used

Sequence

Any sequence of “G s n” and “B t n”

- “G” and “B” may be commands, methods, or events
- “s”, “t”, and “n” may be arguments, parameters, or values
- “s”, “t”, “and “n” may be literals or a set of values

Test Cases from Grammar

- A test case can be a sequence of strings that satisfies the regular expression
- Example

(G s n | B t n)*

Suppose G and B are commands "G" and "B" and s, t, and n are numbers

G 25 08.01.90
B 21 06.27.94
G 21 11.21.94
B 12 01.09.03

Recognizer ("parsing")

- Is a string (or test input) in the grammar?
- Useful for input validation

Generator

- Given a grammar, derive strings in the grammar

Backus-Naur-Form (BNF) Grammars

- Although regular expressions are sometimes sufficient, a more expressive grammar is often used

Start symbol

Stream	::=	action*
action	::=	actG | actB
actG	::=	"G" s n
actB	::=	"B" t n
s	::=	digit$^{1-3}$
t	::=	digit$^{1-3}$
n	::=	digit^{2} "." digit^{2} "." digit^{2}
digit	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Production rules
Possible rewriting of a given nonterminal

Non-terminal symbols

Terminal symbol
Everything in the quotes

More Example: BNF Grammar

- Simple grammar for a toy language of arithmetic expressions in BNF notation

```
expr    ::=  id | num | expr op expr
id      ::=  letter | letter id
num     ::=  digit | digit num
op      ::=  "+" | "-" | "*" | "/"
letter  ::=  "a" | "b" | "c" | … | "z"
digit   ::=  "0" | "1" | "2" | "3" | … | "9"
```

Example: Derivations

```
expr    ::=  id | num | expr op expr
id      ::=  letter | letter id
num     ::=  digit | digit num
op      ::=  "+" | "-" | "*" | "/"
letter  ::=  "a" | "b" | "c" | … | "z"
digit   ::=  "0" | "1" | "2" | "3" | … | "9"
```

a

expr => id => letter => "a"

49

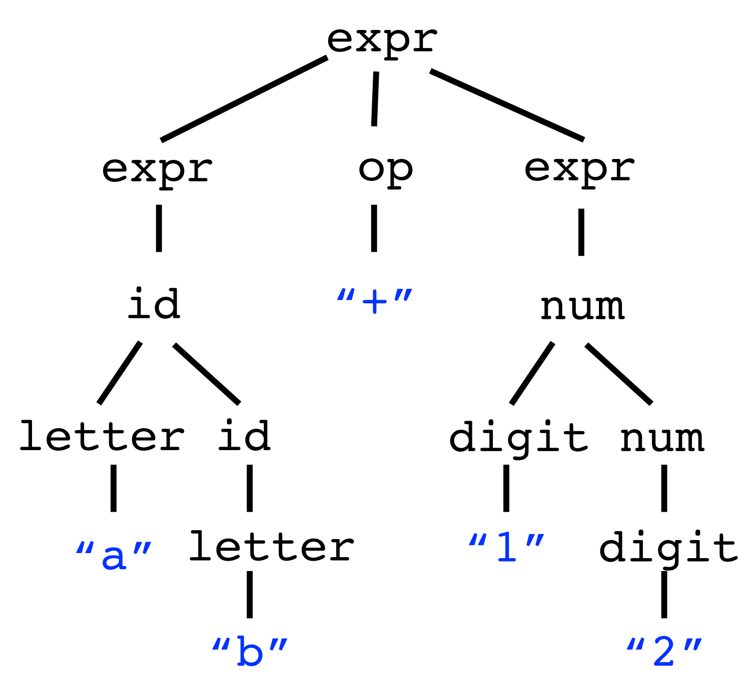
expr => num => digit num => "4" num
=> "4" digit => "4" "9"

ab+12

expr => expr or expr => expr "+" expr
=> … => "a" "b" "+" "1" "2"

syntax tree for

ab+12



Which derivation should be used → leads to how criteria are defined

Grammar Coverage Criteria

- Terminal Symbol Coverage (TSC) — Node Coverage
  - TR contains each terminal in the grammar
  - One test case per terminal
- Production Coverage (PDC) — Edge Coverage
  - TR contains each production rule in the grammar
  - One test case per production (hence PDC subsumes TSC)
- Derivation Coverage (DC) — Complete Path Coverage
  - TR contains every possible derivation of the grammar
  - One test case per derivation
  - Not practical – TR usually infinite
  - When applicable, DC subsumes PDC

Example: TSC

Imagine you are testing a parser or interpreter for the example toy language. Define a test set (i.e., a set of grammar derivations) that satisfies TSC

expr	::=	id | num | expr op expr
id	::=	letter | letter id
num	::=	digit | digit num
op	::=	"+" | "-" | "*" | "/"
letter	::=	"a" | "b" | "c" | … | "z"
digit	::=	"0" | "1" | "2" | "3" | … | "9"

Terminal Symbol Coverage (TSC)

- TR contains each terminal in the grammar
- One test case per terminal

Tests for TSC

Number of tests is bounded by the number of terminal symbols

Need 40 tests

- 26 tests: a, b, …, z
- 10 tests: 0, 1, …, 9
- 4 tests: +, -, *, /

Example: PDC

Imagine you are testing a parser or interpreter for the example toy language. Define a test set (i.e., a set of grammar derivations) that satisfies PDC

expr	::=	id | num | expr op expr
id	::=	letter | letter id
num	::=	digit | digit num
op	::=	"+" | "-" | "*" | "/"
letter	::=	"a" | "b" | "c" | … | "z"
digit	::=	"0" | "1" | "2" | "3" | … | "9"

Production Coverage (PDC)

- TR contains each production rule in the grammar
- One test case per production (hence PDC subsumes TSC)

Tests for PDC

Need 47 tests:

- 40 tests that satisfy TSC
  - 4 for op, 26 for letter,
  - 10 for digit
- Additional 7 tests
  - expr ::=id
  - expr ::= num
  - expr ::= expr op expr
  - id ::= letter
  - id ::= letter id
  - num ::= digit
  - num ::= digit num

Example: DC

Imagine you are testing a parser or interpreter for the example toy language. Define a test set (i.e., a set of grammar derivations) that satisfies DC

expr	::=	id | num | expr op expr
id	::=	letter | letter id
num	::=	digit | digit num
op	::=	"+" | "-" | "*" | "/"
letter	::=	"a" | "b" | "c" | … | "z"
digit	::=	"0" | "1" | "2" | "3" | … | "9"

Derivation Coverage (DC)

- TR contains every possible derivation of the grammar
- One test case per derivation

Tests for DC

- The number of tests depends on details of the program
- For this example:
  - Infinite due to

    id ::= letter id

    num ::= digit num

    expr ::= expr op expr

Mutation Testing

- A process of changing the software artifact based on well defined rules

  Mutation operators: Rules that specify syntactic variations of strings generated from a grammar

- Rules are defined on syntactic descriptions

  Grammars

- We perform mutation analysis when we want to make systematic changes, resulting in variations of a valid string

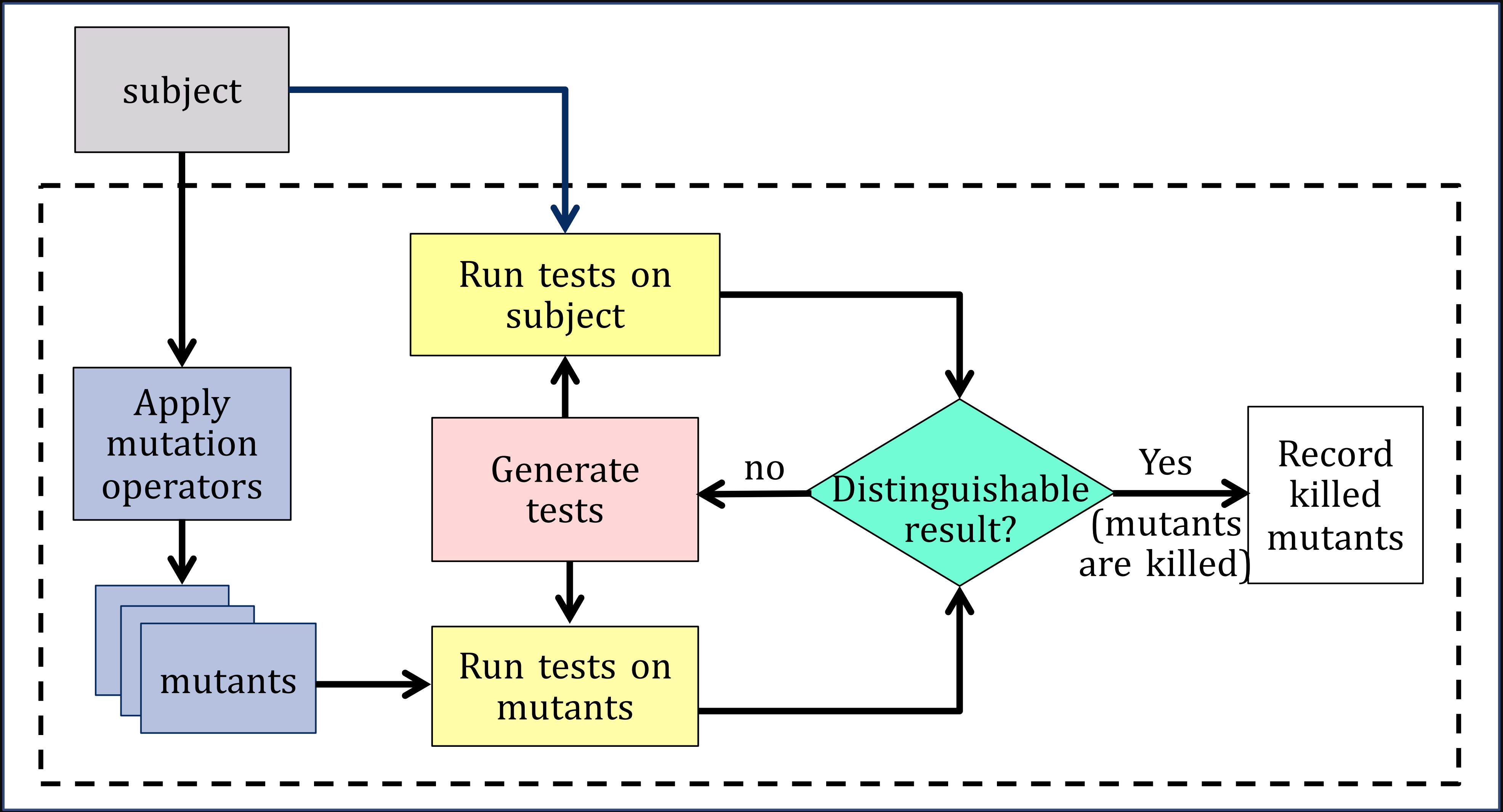
  Mutants: Result of one application of a mutation operator

- We can mutate the syntax or objects developed from the syntax

  Grammar

  Ground strings (Strings in the grammar)

Underlying Concept: Mutation Testing



Mutants and Ground Strings

- Mutation operators
  - The key to mutation testing is the design of the mutation operators
  - Well designed operators lead to powerful testing
- Sometimes mutant strings are based on ground strings
- Sometimes they are derived directly from the grammar
  - Ground strings are used for valid tests
  - Invalid tests do not need ground string

Example: Valid and Invalid Mutants

Stream	::=	action*
action	::=	actG | actB
actG	::=	"G" s n
actB	::=	"B" t n
s	::=	$digit^{1-3}$
t	::=	$digit^{1-3}$
n	::=	$digit^2$ "." $digit^2$ "." $digit^2$
digit	::=	"0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"

Valid Mutants

Ground Strings	Mutants
G 25 08.01.90	**B** 25 08.01.90
B 21 06.27.94	B **4**1 06.27.94

Invalid Mutants

2 25 08.01.90

B 21 06.27.**9**

Grammar-based Mutation Coverage Criteria

- Coverage is defined in terms of killing mutants
- Mutation score $= \frac{number\ killed\ mutants}{total\ \ number\ non{-}equivalent\ mutants}$
- Mutation Coverage (MC)
  - TR contains exactly one requirement to kill each mutant
- Mutation Operator Coverage (MOC)
  - For each mutation operator, TR contains exactly one requirement to create a mutant using that operator
- Mutation Production Coverage (MPC)
  - For each mutation operator, TR contains several requirements to create a mutant that includes every product that can be mutated by that operator

Example Mutation Operators

- Terminal and nonterminal deletion
  - Remove a terminal or nonterminal symbol from a production
- Terminal and nonterminal duplication
  - Duplicate a terminal or nonterminal symbol in a production
- Terminal replacement
  - Replace a terminal with another terminal
- Nonterminal replacement
  - Replace a terminal with another nonterminal

Example

```
Stream   ::=  action*
action   ::=  actG | actB
actG     ::=  "G" s n
actB     ::=  "B" t n
s        ::=  digit1-3
t        ::=  digit1-3
n        ::=  digit2 "." digit2 "." digit2
digit    ::=  "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Ground String

G 25 08.01.90
B 21 06.27.94

Mutation Operators

1. Exchange actG and actB
2. Replace digits with other digits

Mutants using MOC

B 25 08.01.90
B 2**4** 06.27.94

Mutants using MPC

B 25 08.01.90	**G** 21 06.27.94
G **1**5 08.01.90	B 2**2** 06.27.94
G **3**5 08.01.90	B 2**3** 06.27.94
G **4**5 08.01.90	B 2**4** 06.27.94
…	…

Summary

- The number of test requirements for mutation depends
  - The syntax of the artifact being mutated
  - The mutation operators
- Mutation testing is very difficult (and time consuming) to apply by hand
- Mutation testing is very effective – considered the "gold standard" of testing
- Mutation testing is often used to evaluate other criteria