

EE360T/382V Software Testing

khurshid@ece.utexas.edu

April 8, 2020

Overview

Today

- Continue Chapter 5: Syntax-based testing

Next class – continue Chapter 5

Read: Sections 5.4 – 5.5

EE360T/382V Software Testing

khurshid@ece.utexas.edu

Syntax-based testing (Chapter 5)*

*Introduction to Software Testing by Ammann and Offutt

Chapter 5: Outline

Syntax-based coverage criteria

- Using a grammar (or regular expression) to specify test inputs
- Basics of mutation

Program-based grammars

Integration and object-oriented testing

Specification-based grammars

Input space grammars

Background (1)*

Language – set of strings

String – finite sequence of *symbols* (taken from a finite *alphabet*)

Examples:

- Java language – set of all strings that are valid Java programs
- Language of primes – set of all decimal-digit strings that are prime numbers
- Language of Java keywords – {“abstract”, “assert”, “boolean”, “break”, ... }

*Appel: *Modern Compiler Implementation in Java*

Background (2)*

Regular expression – defines a language using a sequence of

- Basic symbols, e.g., $\mathbf{a} = \{ \text{"a"} \}$
- Alternation ($|$), e.g., $\mathbf{a | b} = \{ \text{"a"}, \text{"b"} \}$
- Concatenation ($.$), e.g., $\mathbf{(a | b) . a} = \{ \text{"aa"}, \text{"ba"} \}$
- Epsilon (ϵ) – the language $\{ \text{""} \}$
 - $\mathbf{(a . b) | \epsilon} = \{ \text{""}, \text{"ab"} \}$
- Repetition ($*$) – intuitively, 0+ repetitions
 - $\mathbf{a^*} = \{ \text{""}, \text{"a"}, \text{"aa"}, \text{"aaa"}, \dots \}$
 - $\mathbf{((a | b) . a)^*} = \{ \text{""}, \text{"aa"}, \text{"ba"}, \text{"aaaa"}, \text{"aaba"}, \text{"baaa"}, \text{"baba"}, \text{"aaaaaa"}, \dots \}$

*Appel: *Modern Compiler Implementation in Java*

Example suite – regular expression

Consider testing a container class, say SLList

- Default constructor
- add(int x)
- remove(int x)

Regular expression **((add . 0) | (remove . 0))*** gives an *abstract* representation of a (very large) test suite

```
SLList l = new SLList();
```

```
SLList l = new SLList();  
l.add(0);
```

```
SLList l = new SLList();  
l.remove(0);
```

```
SLList l = new SLList();  
l.add(0);  
l.remove(0);
```

```
SLList l = new SLList();  
l.remove(0);  
l.add(0);
```

```
SLList l = new SLList();  
l.add(0);  
l.add(0);
```

```
SLList l = new SLList();  
l.remove(0);  
l.remove(0);
```

...

Background (3)*

Context-free grammar (BNF) – defines a language using a set of *productions* of the form $sym_0 \rightarrow sym_1 \dots sym_k$

- sym_0 is a **non-terminal**
- Each sym_1, \dots, sym_k is **terminal** (i.e., a basic symbol) or non-terminal
- One symbol is distinguished as the **start symbol**
- ‘|’ indicates choice
- sym^* – 0 or more repetitions of sym
- sym^+ – 1 or more repetitions
- sym^k – exactly k repetitions
- sym^{m-n} – at least m and at most n repetitions

*Appel: *Modern Compiler Implementation in Java*

Example grammar

$S \rightarrow M$

$M \rightarrow I N$

$I \rightarrow \text{add} \mid \text{remove}$

$N \rightarrow D^{1-3}$

$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$

Example string in the language: “add 0”

Example strings not in the language

- “add -1”
- “add 1 add 1”

Two basic uses of grammars

Recognizers – decide if the given string is in the language

- Classical use, e.g., in parsing

Generators – create strings that are in the language

- A use in testing is test input generation
- Example generation (*derivation*)

$S \rightarrow M$ // begin with the start symbol;
 $\rightarrow I N$ // repeatedly replace a non-
 $\rightarrow \text{add } N$ // terminal with its RHS;
 $\rightarrow \text{add } D^{1-3}$ // end when only terminals are
 $\rightarrow \text{add } D$ // left
 $\rightarrow \text{add } 0$

BNF Coverage criteria

Terminal symbol coverage (TSC) – TR contains each terminal in the grammar

- $\#tests \leq \#terminals$, e.g., 12 for our example

Production coverage (PDC) – TR contains each production in the grammar

- $\#tests \leq \#productions$, e.g., 17 for our example
- PDC subsumes TSC

Derivation coverage (DC) – TR contains every string that can be derived from the grammar

- Typically, DC is impractical to use
- $2 * (10 + 100 + 1000)$ tests for our example

Mutation to generate invalid inputs

Using a grammar as a generator allows generating strings that are in the language, i.e., *valid* inputs

Sometimes *invalid* inputs are needed, e.g., to check exception handling behavior or observe failures

Invalid inputs can be created using **mutation**, i.e., (syntactic) modification – the focus of this chapter

Two simple ways to create mutants (valid or invalid):

- Mutate symbols in a ground string
 - E.g., “**add** 0” “**remove** 0”
- Mutate grammar and derive ground strings
 - E.g., “ $I \rightarrow \text{add} \mid \text{remove}$ ” “ $I \rightarrow \text{add} \mid \text{delete}$ ”

Basics of mutation

Assume grammar G defines language L

Ground string – string in L

Mutation operator – rule that specifies (syntactic) variations of strings generated from a grammar

Mutant – result of one application of a mut. operator

- Mutant may be in L (*valid*) or not in L (*invalid*)

Mutation can be used in various ways, e.g.:

- Mutate inputs to programs
 - Check program behaviors on invalid inputs
- Mutate programs themselves – **mutation testing**
 - Evaluate quality of test suites

5.2 Program-based grammars

Grammars that represent programming languages

BNF coverage criteria have been used to generate programs to test compilers

- Specialized application (not discussed here)

Mutation testing has been applied in the context of several languages

- Applying it to Java is a focus of this chapter
 - Ground string – program
 - Mutation operators, e.g., replace ‘+’ ‘-’, create programs that compile, i.e., are valid strings
 - **Mutation-adequate** test suite distinguishes a program from its mutants
 - Such suites likely find (real) program faults

6 example mutants

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    if (b < a)  
    {  
        minVal = b;  
    }  
    return minVal;  
}
```

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    Δ1    minVal = b;  
    if (b < a)  
    Δ2    if (b > a)  
    Δ3    if (b < minVal)  
    {  
        minVal = b;  
    Δ4        fail();  
    Δ5        minVal = a;  
    Δ6        minVal = failOnZero(b);  
    }  
    return minVal;  
}
```

6 example mutants

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    if (b < a)  
    {  
        minVal = b;  
    }  
    return minVal;  
}
```

Δ1, Δ3, Δ5: variable replacement

Δ2: relational op. replacement

Δ4: unconditional failure

Δ6: conditional failure

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    Δ1    minVal = b;  
    if (b < a)  
    Δ2    if (b > a)  
    Δ3    if (b < minVal)  
    {  
        minVal = b;  
    Δ4    fail();  
    Δ5    minVal = a;  
    Δ6    minVal = failOnZero(b);  
    }  
    return minVal;  
}
```


Distinguishing a program from a mutant

Recall the reachability-infection-propagation (RIP) model for failure from Chapter 1:

- Reachability: the test case executes the fault
- Infection: the execution of the fault leads to an erroneous program state
- Propagation: the erroneous state leads to incorrect output

View the mutant m as a (injected) fault in program p

RIP model: to show a behavioral difference between p and m , the test must satisfy reachability and infection, and may also satisfy propagation

Mutant killing

Definition: **strongly killing mutants** – given a mutant m for a program p and a test t , t is said to *strongly kill* m if and only if the output of t on p is different from the output of t on m

Definition: **weakly killing mutants** – given a mutant m that modifies a location l in a program p , and a test t , t is said to *weakly kill* m if and only if the state of the execution of p on t is different from the state of the execution of m on t immediately after location l

Coverage criteria – mutation testing

Let M be the set of mutants of program p

Strong mutation coverage (SMC) – for each m in M , TR contains exactly one requirement, to strongly kill m

Weak mutation coverage (WMC) – for each m in M , TR contains exactly one requirement, to weakly kill m

$$\text{mutation score} = \frac{|\text{mutants killed}|}{|\text{mutants}|}$$

- Consider non-equivalent mutants only (if possible)

Weak mutation example

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    Δ1 minVal = b;  
    if (b < a)  
    {  
        minVal = b;  
    }  
    return minVal;  
}
```

Reachability: true

Infection: $a \neq b$

Propagation: $b < a$ is false
// skip the next assignment

Full test specification to kill
Mutant 1

- $true \wedge a \neq b \wedge \neg(b < a)$,
i.e., $a \neq b \wedge b \geq a$,
i.e., $b > a$

$(a = 5, b = 3)$ will weakly (but
not strongly) kill Mutant 1

Equivalent mutant example

```
static int min(int a, int b) {  
    int minVal;  
    minVal = a;  
    if (b < a)  
Δ3 if (b < minVal)  
    {  
        minVal = b;  
    }  
    return minVal;  
}
```

Infection:

$(b < a) \neq (b < \text{minVal})$

But the previous statement assigns a to minVal

Substituting, we get
 $(b < a) \neq (b < a)$, i.e., a contradiction

Thus, no input can kill this mutant

Testing programs with mutation

Insight: in practice, if the software contains a fault, there will usually be a set of mutants that can only be killed by a test case that also detects the fault

Approach: given a program p

1. Create mutants of p
2. Remove redundant mutants (if feasible)
3. Generate a test suite for p
4. Run each test on p and its mutants to check mutant killing
5. Compute the mutation score for the test suite
6. Check p 's outputs for tests that kill some mutant(s)

Designing mutation operators

Two common strategies for operator design:

- Mimic developer mistakes, e.g., '`<`' '`>`'
- Follow common heuristics, e.g., *failOnZero(...)* uses the heuristic “evaluate expression to 0”

Having more mutation operators means more mutants

Two common ways to control number of mutants

- Randomly sample from total mutants
- Only use *effective* mutation operators
 - Subset E of mutation operators O is **effective** if tests that kill mutants created by E also kill mutants created by $O - E$ with high probability

Mutation operators for Java (1)

ABS – absolute value insertion

- *abs()*, *negAbs()*, *failOnZero()*

AOR – arithmetic operator replacement

- *+*, *-*, ***, */*, *%*, remove an operand/operator

ROR – relational operator replacement

- *>*, *>=*, *<*, *<=*, *==*, *!=*; *false*, *true*

COR – conditional operator replacement

- *&&*, *||*, *&*, *|*, *^*; *false*, *true*

SOR – shift operator replacement

LOR – logical operator replacement

Mutation operators for Java (2)

ASR – assignment operator replacement

- $a = b, a += b, a -= b, a *= b, a /= b, a \% = b, a << = b, a >> = b, a >>> = b, a \& = b, a |= b, \text{ or } a \wedge = b$

UOI – unary operator insertion

- Arithmetic '+', '-', conditional '!', logical '~'

UOD – unary operator deletion

SVR – scalar variable replacement

- " $x = a * b$ " \rightarrow
" $x = a * a$ ", " $a = a * b$ ", " $x = x * b$ ", " $x = a * x$ ",
" $x = b * b$ ", or " $b = a * b$ "

FSR – failure statement replacement

5.3 Integration and OO testing

Integration mutation focuses on component connections

IPVR – integration parameter variable replacement

- “ $m(x)$ ” \rightarrow “ $m(y)$ ”

IUOI – integration unary operator insertion

IPEX – integration parameter exchange

- “ $m(a, b)$ ” \rightarrow “ $m(b, a)$ ”

IMCD – integration method call deletion

- replace call with constant if return value used

IREM – integration return expression modification

- Apply *UOI* or *AOR* to return expression

OO mutation operators focus on OO features

5.5 Input space grammars

Recall we can use a BNF grammar to describe inputs

Inputs for a class of programs (e.g., web services) can be defined using the eXtensible Markup Language (XML)

- XML *schemas* describe a form of grammar

Recall also mutation allows testing with invalid inputs

- When mutating grammars, the mutants themselves are tests
 - Killing mutants does not apply

Mutation operators for grammars

Nonterminal replacement – every nonterminal in a production is replaced by other nonterminals

Terminal replacement – every terminal in a production is replaced by other terminals

Terminal and nonterminal deletion – every terminal and nonterminal in a production is deleted

Terminal and nonterminal duplication – every terminal and nonterminal in a production is duplicated

Exercise

Recall our example grammar

$$S \rightarrow M$$

$$M \rightarrow I N$$

$$I \rightarrow \text{add} \mid \text{remove}$$

$$N \rightarrow D^{1-3}$$

$$D \rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Show example applications of mutation operators for grammars

?/!