UT ECE | THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

EE360T/382V Software Testing

khurshid@ece.utexas.edu

March 5, 2018

Overview

Last class – completed Chapter 4

Today

- Start Chapter 5 – Syntax-based testing

Next class – continue Chapter 5

Read: Sections 5.1 – 5.3

Reminder – Problem Set 3 due: 3/9

UT ECE | THE UNIVERSITY OF TEXAS AT AUSTIN
DEPARTMENT OF ELECTRICAL & COMPUTER ENGINEERING

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., **a** = { “a” }
- Alternation (**|**), e.g., **a | b** = { “a”, “b” }
- Concatenation (**.**), e.g., **(a | b) . a** = { “aa”, “ba” }
- Epsilon (**ϵ**) – the language { “” }
  - **(a . b) | ϵ** = {“”, “ab”}
- Repetition (*) – intuitively, 0+ repetitions
  - **a*** = {“”, “a”, “aa”, “aaa”, … }
  - **((a | b) . a)*** = {“”, “aa”, “ba”, “aaaa”, “aaba”, “baaa”, “baba”, “aaaaaa”, … }

**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 $sym_0 \rightarrow sym_1 \ldots sym_k$

- sym_0 is a non-terminal
- Each $sym_1, \ldots, sym_k$ 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

$S \rightarrow M$

$M \rightarrow I\ N$

$I \rightarrow$ add | 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*)

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

BNF Coverage criteria

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

- #tests ≤ #terminals, e.g., 12 for our example

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

- #tests ≤ #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* → add | remove” → “*I* → add | 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

?/!