L1: Reg Expressions CFGs

- We consider the syntax of the software artifact.
  - BNF grammars for programming languages, modelling languages for design models like UML diagrams and FSMs, XML for inputs and specifications, etc.
- Syntax can be used to generate artifacts that are valid and those that are invalid.
- The structures/artifacts that we generate are sometimes test cases but, most of the time they are not. They are used to generate test cases.
- Coverage criteria based on grammars and regular expressions.
- Mutation testing.
- Mutation testing applied to test source code, design elements, XML inputs etc.

The syntax of many programming language is given in three levels:

- Words: The lexical level, defines how characters form tokens. Generally specified using regular expressions.
- Phrases: The grammar level, determines how tokens form phrases. Generally specified using (deterministic) context-free grammars in Backus-Naur form.
- Context: Deals with types of variables, what they refer to etc. Generally specified using context-sensitive grammars.
- Syntax of regular expressions over an alphabet A:

$$r ::= \emptyset \mid a \mid r + r \mid r \cdot r \mid r^*$$

where $a \in A$.

- Semantics: Associates a language of words or strings $L(r) \subseteq A^*$ with a regular expression r.

$$\begin{array}{lcl} L(\emptyset) & = & \{\} \\ L(a) & = & \{a\} \\ L(r + r') & = & L(r) \cup L(r') \\ L(r \cdot r') & = & L(r) \cdot L(r') \\ L(r^*) & = & L(r)^*. \end{array}$$

Expressions built from a, b, ϵ, using operators $+$, $\cdot$, and $*$.

- $(a^* + b^*) \cdot c$
  Strings of only a's or only b's, followed by a c.
- $(a + b)^* abb (a + b)^*$
  Strings that contain abb as a sub-word.
- $(a + b)^* b (a + b)(a + b)$
  Strings where the 3rd last letter is a b.
- $(ab^*a)^* b^*$
  Strings of b's that (optionally) begin and end with an a, followed by a string of zero or more b's.

CFG G_1

$$\begin{array}{lcl} S & \to & aX \\ X & \to & aX \\ X & \to & bX \\ X & \to & b \end{array}$$

Derivation of a string: Begin with S and keep rewriting the current string by replacing a non-terminal by its RHS in a production of the grammar.
Example derivation:

$$S \Rightarrow aX \Rightarrow abX \Rightarrow abb.$$

Language defined by G, written $L(G)$, is the set of all terminal strings that can be generated by G.
What is the language defined by G_1 above? $a(a+b)^*b$.

A Context-Free Grammar (CFG) is of the form

$$G = (N, A, S, P)$$

where

- N is a finite set of non-terminal symbols
- A is a finite set of terminal symbols.
- $S \in N$ is the start non-terminal symbol.
- P is a finite subset of $N \times (N \cup A)^*$, called the set of productions or rules. Productions are written as

$$X \to \alpha.$$

- "α derives β in 0 or more steps": $\alpha \Rightarrow_G^* \beta$.
- First define $\alpha \xrightarrow{n} \beta$ inductively:
  - $\alpha \xrightarrow{1} \beta$ iff α is of the form $\alpha_1 X \alpha_2$ and $X \to \gamma$ is a production in P, and $\beta = \alpha_1 \gamma \alpha_2$.
  - $\alpha \xrightarrow{n+1} \beta$ iff there exists γ such that $\alpha \xrightarrow{n} \gamma$ and $\gamma \xrightarrow{1} \beta$.
- Sentential form of G: any $\alpha \in (N \cup A)^*$ such that $S \Rightarrow_G^* \alpha$.
- Language defined by G:

$$L(G) = \{w \in A^* \mid S \Rightarrow_G^* w\}.$$

- $L \subseteq A^*$ is called a Context-Free Language (CFL) if there is a CFG G such that $L = L(G)$.

CFG G_1

$$\begin{array}{lcl} S & \to & aXa \\ S & \to & bXb \\ X & \to & aXa \\ X & \to & bXb \\ X & \to & c \end{array}$$

The language generated by the above grammar is $L = \{w \cdot c \cdot w^R \mid w \in (a+b)^*\}$.

- Although regular expressions are sometimes sufficient, a more expressive grammar is often used.
- Also, regular expressions can themselves be written as grammars.
- We will deal with coverage criteria for grammars.

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

Some of the strings generated by the grammar are G 17 08.01.90, B 13 06.27.94 etc.

- Terminal Symbol Coverage (TSC): TR contains each terminal symbol t in the grammar G.
- Production Coverage (PC): TR contains each production p in the grammar G.
- Derivation Coverage (DC): TR contains every possible string that can be derived from the grammar G.
- PDC subsumes TSC, if we cover every production, we cover every terminal.
- DC is an impractical/infeasible coverage criterion. The number of derivations is infinite for many useful grammars.
- These coverage criteria consider generating strings that are members of the language generated by the grammar. In syntax based testing, it is useful to generate strings that are *not* in the grammar too. This will be the focus of next lecture.

L2: Mutation Testing

- Mutation testing involves making syntactically valid changes to a software artifact and then testing the artifact.
- We consider grammars of software artifacts again. Grammars generate valid strings.
- We use derivations in grammars to generate invalid strings too.
- Testing based on these valid and invalid strings is called mutation testing.
- Mutation testing can be applied to almost all software artifacts: code, design, inputs and requirements.
- Ground string: A string that is in the grammar.
- Mutation operator: A rule that specifies syntactic variations of strings generated from a grammar.
- Mutant: The result of *one* application of a mutation operator.
- The ground string could be a program specified using a grammar, in an input format specified using a mark-up language (which has a grammar) etc.
- Mutation operators can also be applied to a grammar, or dynamically during a derivation.
- Appropriately designed mutation operators can be applied to mutate a software artifact and test it.
- Sometimes, ground strings could be the implicit result of *not* applying a mutation operator.
  - When we mutate the inputs to a program, the program stays the same, only the inputs are changed to check if the program responds as expected to invalid inputs.

Consider the grammar that we saw in the last lecture:

```
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
```

- Strings generated by the grammar: G 17 08.01.90, B 13 06.27.94
- Two valid mutants: B 17 08.01.90, G 43 08.01.90
- Two invalid mutants: 12 17 08.01.90, G 23 08.01

- For a given artifact, let M be the set of mutants, each mutant $m \in M$ will lead to a test requirement.
- The testing goal in mutation testing is to kill the mutants by causing the mutant to produce a different output.
- Given a mutant $m \in M$ for a derivation D and a test t, t is said to kill m iff the output of t on D is different from the output of t on m.
- The derivation D could be represented by the complete list of productions followed, or simply be the final string.
- Begin with the program (ground string).
- Apply one or more suitable mutation operators (mutant).
  - Mutants must be programs that compile.
  - Mutants are *not* tests.
- Write tests to kill the mutant.

Six different mutations of the `Min` method. Results in six different programs, each with one mutation.

```
int Min(int A, int B)
{
   int minVal;
   minVal = A;
   if (B<A)
   {
      minVal = B;
   }
   return(minVal);
}
```

```
int Min(int A, int B)
{
   int minVal;
   minVal = A;
△1 minVal = B;
   if (B<A)
△2 if (B>A)
△3 if (B<minVal)
   {
      minVal = B;
△4    Bomb();
△5    minVal = A;
△6    minVal = failOnZero(B);
   }
   return(minVal);
}
```

- Mutation Coverage (MC): For each mutant $m \in M$, TR contains exactly one requirement, to kill m.
- The coverage in mutation equates to killing the mutants.
- The amount of coverage is usually written as a percent of mutants killed and is called the mutation score.
- Higher mutation score doesn't mean more effective testing.
  - When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct.
  - Mutation Operator Coverage (MOC): For each mutation operator, TR contains exactly one requirement, to create a mutated string m that is derived using the mutation operator.
  - Mutation Production Coverage (MPC): For each mutation operator, and each production that the operator can be applied to, TR contains the requirement to create a mutated string from that production.
- Number of test requirements for mutation is somewhat difficult to quantify. It heavily depends on the sytnax of the software artifact and the considered mutation operators.
- Exhaustive mutation testing is never done, operators are available exhaustively to choose from.
- Typically, mutation testing yields more requirements than other criteria.
- Mutation testing is difficult to apply by hand, automation is also complicated.

	For programs	Integration	Specifications	Input space
BNF grammar		–		
Summary	Programming languages Compilers		Algebraic specifications	Input languages (including XML) Input space testing
Mutation				
Summary	Programs Mutates programs	Programs Tests integration	FSMs Model checking	Input languages like XML Error checking

- Exhaustive mutation operators are available for several programming languages to be used for unit and integration testing.
- The goal of mutation operators is to mimic programmer mistakes.
- Mutation operators have to be selected carefully to strengthen the quality of mutation testing.
- Test cases are effective if they result in the mutated program exhibiting a behavior different from the original program.
- Several different mutants are defined in the testing literature to precisely capture the quality of mutation.
  - Stillborn mutant: Mutants of a program result in *invalid* programs that cannot even be compiled. Such mutants should not be generated.
  - Trivial mutant: A mutant that can be killed by almost any test case.
  - Equivalent mutant: Mutants that are functionally equivalent to a given program. No test case can kill them.
  - Dead mutant: Mutants that are valid and can be killed by a test case. These are the only useful mutants for testing.
- Strongly killing mutants: Given a mutant $m \in M$ for a ground string program P and a test t, t is said to strongly kill m iff the output of t on P is different from the output of t on m.
- Strong Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement to strongly kill m.
- Weakly killing mutants: Given a mutant $m \in M$ that modifies a location l in a program P, and a test t, t is said to weakly kill m iff the state of the execution of P on t is different from the state of execution of m immediately after l.
- Weak Mutation Coverage (SMC): For each $m \in M$, TR contains exactly one requirement to weakly kill m.

```
1  boolean isEven(int X)
2  {
3     if(X < 0)
4        X = 0-X;
△4       X = 0;
5     if(float)(X/2) == ((float)X)/2.0
6        return(true);
7     else
8        return(false);
9  }
```

- The mutant in line 4 is an example of weak killing.
  - Reachability: $(X < 0)$.
  - Infection: $(X \neq 0)$.
- Consider the test case X=-6. Value of X after line 4:
  - In the original program: 6
  - In the mutated program: 0.
- Since 6 and 0 are both even, the decision at line 5 will return true for both the versions, so propagation is not satisfied.
- For strong killing, we need the test case to be an odd, negative integer.

L3: Mutant Operators Source Code

- Effective mutation operators: If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, \ldots\}$ also kill mutants created by all remaining mutation operators with very high probability, then O defines an effective set of mutation operators.
- It is difficult to find an effective set of mutation operators for a given program.
- Empirical studies have indicated that mutation operators that insert unary operators and those that modify unary and binary operators will be effective.

We provide a list of program level mutation operators that deal with unary and binary, relational and arithmetic operators. They can be applied to many programming languages.

- Absolute Value Insertion: Each arithmetic expression (and sub-expression) is modified by the functions `Abs()`, `negAbs()` and `failOnZero()`.
  - `Abs()` returns the absolute value of the expression.
  - `negAbs()` returns the negative of the absolute value.
  - `failOnZero()` tests whether the value of the expression is zero. If it is, the mutant is killed. Otherwise, execution continues and the value of the expression is returned.
- Arithmetic Operator Replacement: Each occurrence of one of the arithmetic operators +, -, *, /, ** and % is replaced by each of the other operators.
- In addition, each is replaced by the special mutation operators *leftOp*, *rightOp* and *mod*.
  - *leftOp* returns the left operand (the right is ignored).
  - *rightOp* returns the right operand.
  - *mod* computes the remainder when the left operand is divided by the right.
- Relational Operator Replacement: Each occurrence of one of the relational operators ($<, >, \leq, \geq, =, \neq$) is replaced by each of the other operators and by *falseOp* and *trueOp*.
  - *falseOp* always returns false and *trueOp* always returns true.
- Conditional Operator Replacement: Each occurrence of each logical operator (and-&&, or-|||, and with no conditional evaluation- &, or with no conditional evaluation- ||, not equivalent-^) is replaced by each of the other operators. In addition, each is replaced by *falseOp*, *trueOp*, *leftOp* and *rightOp*.
  - *leftOp* returns the left operand (the right is ignored) and *rightOp* returns the right operand (the left is ignored).
- Shift Operator Replacement: Each occurrence of one of the shift operators <<, >> and >>> is replaced by each occurrence of the other operators. In addition, each is replaced by the special mutation operator *leftOp*.
  - *leftOp* returns the left operand unshifted.
- Logical Operator Replacement: Each occurrence of each bitwise logical operator (*bitwise and* (&), *bitwise or* (|) and *exclusive or* (^)) is replaced by each of the other operators. In addition, each is replaced by *leftOp* and *rightOp*.
  - *leftOp* returns the left operand (right is ignored) and *rightOp* returns the right operand (left is ignored).
- Assignment Operator Replacement: Each occurrence of one of the assignment operators (+=, -=, *=, =, %=, &=, |=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators.
- Unary Operator Insertion: Each unary operator (arithmetic +, arithmetic -, conditional , logical ~) is inserted before each expression of the correct type.
  - Unary Operator Deletion: Each unary operator (arithmetic +, arithmetic -, conditional , logical ~) is deleted.
- Scalar variable replacement: Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope.
- Bomb Statement Replacement: Each statement is replaced by a special `Bomb()` function.
- `Bomb()` signals a failure as soon as it is executed, thus requiring the tester to reach each statement.

- Typical coverage criteria impose only a *local* requirement.
  - For e.g., edge coverage requires that each branch in a program be executed.
- Mutation, on the other hand, imposes a *global* requirement in addition to local requirements. It requires that mutant program produce incorrect output.
  - For edge coverage, some mutants can be killed only if each branch is executed *and* the final output of the mutant is incorrect.
- Mutation testing thus imposes *stronger* requirements than the other coverage criteria.
- The problem of mutation testing being global and coverage criteria being local is solved by basing subsumptions on weak mutations.
- Weak mutation means that mutants are *not* equivalent at the infection stage but are equivalent at the propagation stage.
- Hence, weak mutation can be made to subsume edge coverage.
- In general, we consider weak mutation throughout this lecture.
  - Mutation testing subsumes node coverage.
  - The mutation operator that replaces statements with "bombs" yields node coverage.
  - To kill the "bomb" mutant, we have to find test cases that reach each basic block— the same as node coverage.
- Mutation testing subsumes edge coverage.
- The mutation operator to be applied to subsume edge coverage is to replace each logical predicate with both `true` and `false` (Relational operator replacement mutation operator).
- To kill the true mutant, a test case must take the false branch and to kill the false mutant, a test case must take the true branch.
- This forces each branch in the program to be executed, hence, mutation subsumes edge coverage.
  - Predicate coverage for logic is the same as edge coverage for graphs, hence, mutation subsumes predicate coverage.
  - We will show that mutation subsumes clause coverage too.
  - Mutation *does not* subsume combinatorial coverage.
    - Combinatorial coverage requires 2^n requirements for a predicate with n clauses.
    - No single (or combination of) mutation operator(s) can produce 2^n mutants.
- Mutation testing subsumes general active clause coverage criterion.
- ACC requires that each clause in a predicate p evaluate to true and false *and* determine the value of p.
- To kill the mutant $\triangle p(a \rightarrow true)$, we must satisfy infection by causing $p(a \rightarrow true)$ to have a value a different value from a.
- Likewise, to kill $\triangle p(a \rightarrow false)$, we must satisfy infection by causing $p(a \rightarrow false)$ to have a value a different value from a.
- This is exactly GACC; hence, mutation testing subsumes GACC.
- Strong mutation is needed to subsume all-defs coverage.
- We consider only statements that contain variable definitions.
- The mutation applied is to delete such statements.
- This will end up "covering" the statements that define variables.

L4: Mutation Testing vs Other Testing

- Mutation is considered as the strongest test criterion in terms of finding the most faults.
- Mutation subsumes a number of other coverage criteria.
  - Mutation operators can be picked to impose requirements that are identical to a specific coverage criterion.
  - For each specific requirement, a single mutant is created that can be killed only by the test cases that satisfy the requirement.
  - Mutation operators that ensure coverage of a criterion are said to yield the criterion.
- We use mutation operators that are generically applicable while considering subsumption.

L5: Mutation Testing for Integration and Tools

- Integration mutation (sometimes called interface mutation) works by creating mutants on the connections between components.
- Most of the mutations are around method calls, and both the calling (caller) and the called (callee) method are considered.

Integration mutation operators do the following:

- Change a calling method by modifying the values that are sent to a called method.
- Change a calling method by modifying the call.
- Change a called method by modifying the values that enter and leave a method.
  - This should include parameters and variables from a higher scope (class level, package, public etc.)
- Change a called method by modifying statements that return from the method.

Integration mutation 5

The following are the five interface mutation operators that we will define.

- Integration Parameter Variable Replacement (IVPR).
- Integration Unary Operator Insertion (IUOI).
- Integration Parameter Exchange (IPEX).
- Integration Method Call Deletion (IMCD).
- Integration Return Expression Modification (IREM).

- Integration Parameter Variable Replacement (IVPR): Each parameter in a method call is replaced by each other variable of compatible type in the scope of the method call.
  - IPVR does not use variables of an incompatible type because they would be syntactically illegal.
  - In OO languages, this operator replaces primitive type variables as well as objects.
- Integration Unary Operator Insertion (IUOI): Each expression in a method call is modified by inserting all possible unary operators in front and behind it.
  - The unary operators vary by the language and type.
- Integration Parameter Exchange (IPEX): Each parameter in a method call is exchanged with each parameter of compatible type in that method call.
  - For e.g., if a method call is `max(a,b)`, a mutated method call of `max(b,a)` is created.
- Integration Method Call Deletion (IMCD): Each method call is deleted. If the method returns a value and it is used in an expression, the method call is replaced with an appropriate constant value.
  - In Java, the default values should be used for methods that return values of primitive type. If a method returns an object, the method call should be replaced by a call to `new()` on the appropriate class.

remember to typecast in new().

- Integration Return Expression Modification (IREM): Each expression in each return statement in a method is modified by applying the UOI (Unary Operator Insertion) and AOR (Arithmetic Operator Replacement) operators.

Tools for Mutation Testing —

1. pitest.org → For java.
2. Stryker → For JS, C#, Scala
3. Jumble → For Junit tests based OO concepts.