# Chapter 9

# Syntax-based Testing

*Date last generated: September 29, 2014.*

*DRAFT: Prepublication draft, Fall 2014, George Mason University*
*Redistribution is forbidden without the express permission of the authors*

*Colorless green ideas sleep furiously.*

— Noam Chomsky

In previous chapters, we learned how to generate tests from the input space, graphs, and logical expressions. These criteria required reachability (for graphs) and infection (for logical expressions). A fourth major source for test coverage criteria is syntactic descriptions of software artifacts, which allows propagation to be required. As with graphs and logical expressions, several types of artifacts can be used, including source and input requirements.

The essential characteristic of syntax-based testing is that a syntactic description such as a grammar or BNF is used. Chapter 6 discussed how to build a model of the inputs based on some description of the input space. Chapters 7 and 8 discussed how to build graph models and logic models from artifacts such as the program, design descriptions, and specifications. Then test criteria were applied to the models. With syntax-based testing, however, the syntax of the software artifact is used as the model and tests are created from the syntax.

## 9.1 Syntax-based Coverage Criteria

Syntax structures can be used for testing in several ways. We can use the syntax to generate artifacts that are valid (correct syntax), or artifacts that are invalid (incorrect syntax). Sometimes the structures we generate are test cases themselves, and sometimes they are used to help us design test cases. We explore these differences in the subsections of this chapter. As usual, we begin by defining general criteria on syntactic structures, and then make them specific to specific artifacts.

### 9.1.1 Grammar-based Coverage Criteria

It is very common in software engineering to use structures from automata theory to describe the syntax of software artifacts. Programming languages are described in BNF grammar notation, program behavior is described in finite state machines, and allowable inputs to programs are defined by grammars. Regular expressions and context free grammars are especially useful. Consider the regular expression:

$(G\ s\ n \mid B\ t\ n)^*$

The star is a "closure" operator that indicates zero or more occurrences of the expression it modifies. The vertical bar is the "choice" operator, and indicates either choice can be taken. Thus, this regular expression describes any sequence of "$G\ s\ n$" and "$B\ t\ n$." $G$ and $B$ may be commands to a program and $s$, $t$ and $n$ may be arguments, method calls with parameters, or messages with values. The arguments $s$, $t$ and $n$ can be literals or represent a large set of values, for example, numbers or strings.

A test case can be a sequence of strings that satisfy the regular expression. For example, if the arguments are supposed to be numbers, the following may represent one test with four components, two separate tests, three separate tests, or four separate tests:

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

Although regular expressions are sometimes sufficient, a more expressive grammar is often used. The prior example can be refined into a grammar form as follows:

```
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"
```

**Meta Discussion**

*We simplify the syntax a bit in our examples. Specifically, we intentionally omit spaces. More formal treatments are given in formal language textbooks, however that level of formalism is not needed for testing. Details of the syntax will be added when test requirements are refined into executable tests.*

A grammar has a special symbol called the *start symbol*. In this case, the start symbol is `stream`. Symbols in the grammar are either *nonterminals*, which must be rewritten further, or *terminals*, for which no rewriting is possible. In the example, the symbols on the left of the ::= sign are all nonterminals, and everything in quotes is a terminal. Each possible rewriting of a given nonterminal is called a *production* or *rule*. In this grammar, a star superscript means zero or more, a plus superscript means one or more, a numeric superscript indicates the required number of repetitions, and a numeric range $(a - b)$ means there has to be at least $a$ repetitions, and no more than $b$.

Grammars can be used in two ways. A *recognizer*, as defined in Chapter 5, decides whether a given string (or test case) is in the grammar. This is the classical automata theory problem of parsing, and automated tools (such as the venerable `lex` and `yacc`) make the construction of recognizers very easy. Recognizers are extremely useful in testing, because they make it possible to decide if a given test case is in a particular grammar or not. The other use of grammars is to build *generators*, also defined in Chapter 5. A generator derives a string of terminals from the grammar start symbol. In this case, the strings are test inputs. For example, the following derivation results in the test case `G 25 08.01.90`.

```
stream → action^*
       → action action^*
       → actG action^*
       → G s n action^*
       → G digit^(1-3) digit^2 .  digit^2 .  digit^2 action^*
       → G digitdigit digitdigit.digitdigit.digitdigit action^*
       → G 25 08.01.90 action^*
       ⋮
```

The derivation proceeds by systematically replacing the next nonterminal (for example, "`action^*`") with one of its productions. Derivation continues until all nonterminals have been rewritten and only terminal symbols remain. The key to testing is which derivations should be used, and this is how criteria are defined on grammars.

Although many test criteria could be defined, the most common and straightforward are *terminal symbol coverage* and *production coverage*.

CRITERION **9.1 Terminal Symbol Coverage (TSC):** *TR contains each terminal symbol t in the grammar G.*

CRITERION **9.2 Production Coverage (PDC):** *TR contains each production p in the grammar G.*

By now, it should be easy to see that PDC subsumes TSC (if we cover every production, we cover every terminal symbol). Some readers may also note that grammars and graphs

have a natural relationship. Therefore, Terminal Symbol Coverage and Production Coverage can be rewritten to be equivalent to Node Coverage and Edge Coverage on the graph that represents the grammar. Of course, this means that the other graph-based coverage criteria can also be defined on grammars. To our knowledge, neither researchers nor practitioners have taken this step.

The only other related criterion defined here is the impractical one of deriving all possible strings in a graph.

CRITERION **9.3 Derivation Coverage (DC):** *TR contains every possible string that can be derived from the grammar G.*

The number of tests generated by TSC will be bounded by the number of terminal symbols. The `stream` BNF above has 13 terminal symbols: G, B, ., 0, 1, 2, 3, 4, 5, 6, 7, 8, 9. It has 18 productions (note the '|' symbol adds productions, so "`action`" has two productions and "`digit`" has 10). The number of derivations for DC depends on the details of the grammar, but generally can be infinite. If we ignore the first production in the `stream` BNF, we have a finite number of derivable strings. Two possible actions are actG and actB, $s$ and $t$ each has a maximum of three digits with 10 choices, or 1000. The nonterminal $n$ has three sets of two digits with 10 choices apiece, or $10^6$. Altogether, the `stream` grammar can generate $2 * 1000 * 10^6 = 2,000,000,000$ strings. DC is of theoretical interest but is obviously impractical. (A point to remember the next time a tool salesperson or job applicant claims to have done "full string coverage" or "full path coverage.")

TSC, PDC and DC generate test cases that are members of the set of strings defined by the grammar. It is sometimes very helpful to generate test cases that are **not** in the grammar, which is addressed by the criteria in the next subsection.

---

**Exercises, Section 9.1.1.**

1. Consider how often the idea of covering nodes and edges pops up in software testing. Write a short essay to explain this.

2. Just as with graphs, it is possible to generate an infinite number of tests from a grammar. How and what makes this possible?

---

### 9.1.2 Mutation Testing

One of the interesting things that grammars do is describe what an input is *not*. We say that an input is *valid* if it is in the language specified by the grammar, and *invalid* otherwise.

For example, it is quite common to require a program to reject malformed inputs, and this property should clearly be tested, since it is easy for programmers to forget it or get it wrong.

Thus, it is often useful to produce invalid strings from a grammar. It is also helpful in testing to use strings that are *valid* but that follow a different derivation from a pre-existing string. Both of these strings are called *mutants*[1]. This can be done by mutating the grammar, then generating strings, or by mutating values during a derivation.

Mutation can be applied to various artifacts, as discussed in the following subsections. However, it has primarily been used as a program-based testing method, and much of the theory and many of the detailed concepts are specific to program-based mutation. Therefore, a lot more details appear in Section 9.2.2.

Mutation is always based on a set of "mutation operators," which are expressed with respect to a "ground" string.

**Definition 9.1 Ground String:** *A string that is in the grammar.*

**Definition 9.2 Mutation Operator:** *A rule that specifies syntactic variations of strings generated from a grammar.*

**Definition 9.3 Mutant:** *The result of one application of a mutation operator.*

Mutation operators are usually applied to ground strings, but can also be applied to a grammar, or dynamically during a derivation. The notion of a mutation operator is extremely general, and so a very important part of applying mutation to any artifact is the design of suitable mutation operators. A well designed set of operators can result in very powerful testing, but a poorly designed set can result in ineffective tests. For example, a commercial tool that "implements mutation" but that only changes predicates to *true* and *false* would simply be an expensive way to implement branch coverage.

We sometimes have a particular ground string in mind, and sometimes the ground string is simply the implicit result of not applying any mutation operators. For example, we care about the ground string when applying mutation to program statements. The ground string is the sequence of program statements in the program under test, and the mutants are slight syntactic variations of that program. We do not care about the ground string during invalid input testing, when the goal is to see if a program correctly responds to invalid inputs. The ground strings are valid inputs, and variants are the invalid inputs. For example, a valid input might be a transaction request from a correctly logged-in user. The invalid version might be the same transaction request from a user who is not logged in.

Consider the grammar in Section 9.1.1. If the first string shown, `G 25 08.01.90`, is taken as a ground string, two *valid* mutants may be:

```
B 25 08.01.90
G 43 08.01.90
```

[1]There is no relationship between this use of mutation and genetic algorithms, except that both make an analogy to biological mutation. Mutation for testing predated genetic algorithms by decades.

Two *invalid* mutants may be:

```
12 25 08.01.90
G 25 08.01
```

When the ground string does not matter, mutants can be created directly from the grammar by modifying productions during a derivation, using a generator approach as introduced in the previous section. That is, if the ground strings are not of direct interest, they do not need to be explicitly generated.

When applying mutation operators, two issues often come up. First, should more than one mutation operator be applied at the same time to create one mutant? That is, should a mutated string contain one mutated element, or several? Common sense indicates no, and strong experimental and theoretical evidence has been found that indicates we usually only want to mutate one element at a time in program-based mutation. An exception is where so called "subsuming higher order mutants" can be useful; we do not discuss this topic. Another question is should every possible application of a mutation operator to a ground string be considered? This is usually done in program-based mutation. One theoretical reason is that program-based mutation subsumes a number of other test criteria, and if operators are not applied comprehensively, then that subsumption is lost. However, this is not always done when the ground string does not matter, for example, in the case of invalid input testing. This question is explored in more detail in the following application subsections.

Mutation operators have been designed for several programming languages, formal specification languages, BNF grammars, and at least one data definition language (XML). For a given artifact, the set of mutants is $M$ and each mutant $m \in M$ will lead to a test requirement.

When a derivation is mutated to produce valid strings, the testing goal is to "kill" the mutants by causing the mutant to produce different output. More formally, given a mutant $m \in M$ for a derivation $D$ and a test $t$, $t$ is said to *kill* $m$ if and only if the output of $t$ on $D$ is different from the output of $t$ on $m$. The derivation $D$ may be represented by the complete list of productions followed, or it may simply be represented by the final string. For example, in Section 9.2.2, the strings are programs or program components. Coverage is defined in terms of killing mutants.

CRITERION **9.4 Mutation Coverage (MC):** *For each mutant $m \in M$, $TR$ contains exactly one requirement, to kill $m$.*

Thus, coverage in mutation equates to killing the mutants. The amount of coverage is usually written as the ratio of mutants killed over all mutants and called the "*mutation score*."

When a grammar is mutated to produce invalid strings, the testing goal is to run the mutants to see if the behavior is correct. The coverage criterion is therefore simpler, as the mutation operators are the test requirements.

CRITERION **9.5 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.*

CRITERION **9.6 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.*

The number of test requirements for mutation is somewhat difficult to quantify because it depends on the syntax of the artifact as well as the mutation operators. In most situations, mutation yields more test requirements than any other test criterion. Subsequent sections have some data on quantifying specific collections of mutation operators and more details are in the bibliographic notes.

Mutation testing is also difficult to apply by hand, and automation is more complicated than for most other criteria. As a result, mutation is widely considered a "high-end" test criterion, more effective than most but also more expensive. One common use of mutation is as a sort of "gold standard" in experimental studies for comparative evaluation of other test criteria.

The rest of this chapter explores various forms of BNF and mutation testing. The table below summarizes the sections and the characteristics of the various flavors of syntax testing. Whether the use of syntax testing creates valid or invalid tests is noted for both BNF and mutation testing. For mutation testing, we also note whether a ground string is used, whether the mutants are tests or not, and whether mutants are killed.

| | **Program-based** | **Integration** | **Specification-based** | **Input space** |
|---|---|---|---|---|
| **BNF** | 9.2.1 | 9.3.1 | 9.4.1 | 9.5.1 |
| Grammar | Programming languages | No known applications | Algebraic specifications | Input languages, including XML |
| Summary | Compiler testing | | | Input space testing |
| **Mutation** | 9.2.2 | 9.3.2 | 9.4.2 | 9.5.2 |
| Grammar | Programming languages | Programming languages | FSMs | Input languages, including XML |
| Summary | Mutates programs | Tests integration | Uses model-checking | Error checking |
| Ground? | Yes | Yes | Yes | No |
| Valid? | Yes, must compile | Yes, must compile | Yes | No |
| Tests? | Mutants are not tests | Mutants are not tests | Mutants are not tests | Mutants are tests |
| Killing? | Yes | Yes | Yes | No notion of killing |
| Notes | Strong and weak mutants. Subsumes many other techniques. | Includes object-oriented testing | Automatic detection of equivalent mutants | Sometimes the grammar is mutated, then strings are produced |

---

**Exercises, Section 9.1.2.**

1. Define mutation score.

2. How is the mutation score related to coverage from Chapter 5?

3. Consider the stream BNF in Section 9.1.1 and the ground string "`B 21 06.27.94`". Give three valid and three invalid mutants of the string.

---

## 9.2 Program-based Grammars

As with most criteria, syntax-based testing criteria have been applied to programs more than other artifacts. The BNF coverage criteria have been used to generate programs to test compilers. Mutation testing has been applied to methods (unit testing) and to classes (integration testing). Application to classes is discussed in the next section.

### 9.2.1 BNF Grammars for Compilers

The primary purpose of BNF testing for languages has been to generate test suites for compilers. As this is a very specialized application, we choose not to dwell on it in this book. The bibliographic notes section has pointers to the relevant, mostly rather old, literature.

### 9.2.2 Program-based Mutation

Mutation was originally developed for programs and this section has significantly more depth than other sections in this chapter. Program-based mutation uses operators that are defined in terms of the grammar of a particular programming language. We start with a **ground string**, which is a program that is being tested. We then apply mutation operators to create mutants. These mutants must be compilable, so program-based mutation creates **valid** strings. The mutants are not tests, but are used to help us design tests.

Given a ground string program or method, a mutation-adequate test set distinguishes the program from a set of syntactic variations, or mutants, of that program. A simple example of a mutation operator for a program is the *Arithmetic Operation Mutation* operator, which changes an assignment statement like "`x = a + b`" into a variety of alternatives, including "`x = a - b`," "`x = a * b`," and "`x = a / b`." Unless the assignment statement appears in a very strange program, it probably matters which arithmetic operator is used, and a decent test set should be able to distinguish among the various possibilities. It turns out that by careful selection of the mutation operators, a tester can develop very powerful test sets.

Mutation testing is used to help the user strengthen the quality of test data iteratively. Test data are used to evaluate the ground program with the goal of causing each mutant to exhibit different behavior. When this happens, the mutant is considered *dead* and no longer needs to remain in the testing process since the fault that it represents will be detected by the same test that killed it. More importantly, the mutant has satisfied its requirement of identifying a useful test case.

A key to successful use of mutation is the mutation operators, which are designed for each programming, specification, or design language. In program-based mutation, invalid strings are syntactically illegal and would be caught by a compiler. These are called *stillborn* mutants and either should not be generated or should be immediately discarded. A *trivial* mutant can be killed by almost any test case. Some mutants are functionally *equivalent* to the original program. That is, they always produce the same output as the original program, so no test case can kill them. Equivalent mutants represent infeasible test requirements, as discussed in the previous chapters.

We refine the notion of killing and coverage for program-based mutation. These definitions are consistent with the previous section.

**Definition 9.4 Killing Mutants:** *Given a mutant $m \in M$ for a ground string program $P$ and a test $t$, $t$ is said to kill $m$ if and only if the output of $t$ on $P$ is different from the output of $t$ on $m$.*

As said in Section 9.1.2, it is hard to quantify the number of test requirements for mutation. In fact, it depends on the specific set of operators used and the language that the operators are applied to. One of the most widely used mutation systems was Mothra. It generated 44 mutants for the Fortran version of the `Min()` method in Figure 9.1. For most collections of operators, the number of program-based mutants is roughly proportional to the product of the number of references to variables times the number of variables that are declared ($O(Refs * Vars)$). The selective mutation approach mentioned below under "Designing Mutation Operators" eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$). More details are in the bibliographic notes.

Program-based mutation has traditionally been applied to individual statements for unit level testing. Figure 9.1 contains a small Java method with six mutated lines (each preceded by the $\Delta$ symbol). Note that each mutated statement represents a separate program. The mutation operators are defined to satisfy one of two goals. One goal is to mimic typical programmer mistakes, thus trying to ensure that the tests can detect those mistakes. The other goal is to force the tester to create tests that have been found to effectively test software. In Figure 9.1, mutants 1, 3, and 5 replace one variable reference with another, mutant 2 changes a relational operator, and mutant 4 is a special mutation operator that causes a runtime failure as soon as the statement is reached. This forces every statement to be executed, thus achieving statement or node coverage.

Mutant 6 looks unusual, as the operator is intended to force the tester to create an effective test. The *failOnZero()* method is a special mutation operator that causes a failure if the parameter is zero, and does nothing if the parameter is not zero (it returns the value of the parameter). Thus, mutant 6 can be killed only if **B** has the value zero, which forces the tester to follow the time-tested heuristic of causing every variable and expression to have the value of zero.

One point that is sometimes confusing about mutation is how tests are created. When applying program-based mutation, the direct goal of the tester is to kill mutants; an indirect

| Original Method | | With Embedded Mutants |
|---|---|---|
| `int Min (int A, int B)` | | `int Min (int A, int B)` |
| `{` | | `{` |
| `  int minVal;` | | `  int minVal;` |
| `  minVal = A;` | | `  minVal = A;` |
| `  if (B < A)` | Δ1 | `  minVal = B;` |
| `  {` | | `  if (B < A)` |
| `    minVal = B;` | Δ2 | `  if (B > A)` |
| `  }` | Δ3 | `  if (B < minVal)` |
| `  return (minVal);` | | `  {` |
| `} // end Min` | | `    minVal = B;` |
| | Δ4 | `    Bomb();` |
| | Δ5 | `    minVal = A;` |
| | Δ6 | `    minVal = failOnZero (B);` |
| | | `  }` |
| | | `  return (minVal);` |
| | | `} // end Min` |

Figure 9.1: Method Min and six mutants.

goal is to create good tests. Even less directly, the tester wants to find faults. Tests that kill mutants can be found by intuition, or if more rigor is needed, by analyzing the conditions under which a mutant will be killed.

The RIPR fault/failure model was introduced in Chapter 2. Program-based mutations represent a software failure by a mutant, and reachability, infection, and propagation refer to reaching the mutant, the mutant causing the program state to be incorrect, and the eventual output of the program to be incorrect.

*Weak mutation* relaxes the definition of "killing" a mutant to include only reachability and infection, but **not** propagation. Weak mutation checks the internal state of the program immediately after execution of the mutated component (that is, after the expression, statement, or basic block). If the state is incorrect the mutant is killed. This is weaker than standard (or *strong*) mutation because an incorrect state does **not** always propagate to the output. That is, strong mutation may require more tests to satisfy coverage than weak mutation. Experimentation has shown that the difference is very small in most cases.

This difference can be formalized by refining the definition of killing mutants given previously.

**Definition 9.5 Strongly Killing Mutants:** *Given a mutant $m \in M$ 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 $m$.*

CRITERION **9.7 Strong Mutation Coverage (SMC):** *For each $m \in M$, $TR$ contains exactly one requirement, to strongly kill $m$.*

**Definition 9.6 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 $m$ if and only if the state of the execution of $P$ on $t$ is different from the state of the execution of $m$ immediately after $l$.*

CRITERION **9.8 Weak Mutation Coverage (WMC):** *For each $m \in M$, $TR$ contains exactly one requirement, to weakly kill $m$.*

Consider mutant 1 in Figure 9.1. The mutant is on the first statement, thus the reachability condition is always satisfied (*true*). In order to infect, the value of **B** must be different from the value of **A**, which is formalized as $(A \neq B)$. To propagate, the mutated version of **Min** must return an incorrect value. In this case, **Min** must return the value that was assigned in the first statement, which means that the statement inside the **if** block must **not** be executed. That is, $(B < A) = false$. The complete test specification to kill mutant 1 is:

Reachability: $true$
Infection: $A \neq B$
Propagation: $(B < A) = false$
Full Test Specification: $true \wedge (A \neq B) \wedge ((B < A) = false)$
$\equiv (A \neq B) \wedge (B \geq A)$
$\equiv (B > A)$

Thus, the test case value $(A = 5, B = 7)$ should cause mutant 1 to result in a failure. The original method will return the value 5 $(A)$ but the mutated version returns 7.

Mutant 3 is an example of an equivalent mutant. Intuitively, **minVal** and **A** have the same value at that point in the program, so replacing one with the other has no effect. As with mutant 1, the reachability condition is *true*. The infection condition is $(B < A) \neq (B < minVal)$. However, dutiful analysis can reveal the assertion $(minVal = A)$, leading to the combined condition $((B < A) \neq (B < minVal)) \wedge (minVal = A)$. Simplifying by eliminating the inequality $\neq$ gives:

$$(((B < A) \wedge (B \geq minVal)) \vee ((B \geq A) \wedge (B < minVal))) \wedge (minVal = A)$$

Rearranging the terms gives:

$$(((A > B) \wedge (B \geq minVal)) \vee ((A \leq B) \wedge (B < minVal))) \wedge (minVal = A)$$

If $(A > B)$ and $(B \geq minVal)$, then by transitivity, $(A > minVal)$. Applying transitivity to both the first two disjuncts gives:

$$((A > minVal) \vee (A < minVal)) \wedge (minVal = A)$$

Finally, the first disjunct can be reduced to a simple inequality, resulting in the following contradiction:

$$(A \neq minVal) \wedge (minVal = A)$$

The contradiction means that no values exist that can satisfy the conditions, thus the mutant is provably equivalent. In general, detecting equivalent mutants, just like detecting infeasible paths, is an undecidable problem. However, strategies such as algebraic manipulations and program slicing can detect some equivalent mutants.

As a final example, consider the following method, with one mutant shown embedded in statement 4:

```
1     boolean isEven (int X)
2     {
3        if (X < 0)
4           X = 0 - X;
Δ4          X = 0;
5        if (float) (X/2) == ((float) X) / 2.0
6           return (true);
7        else
8           return (false);
9     }
```

The reachability condition for mutant $\Delta 4$ is $(X < 0)$ and the infection condition is $(X \neq 0)$. If the test case `X = -6` is given, then the value of `X` after statement 4 is executed is 6 and the value of `X` after the **mutated** version of statement 4 is executed is 0. Thus, this test satisfies reachability and infection, and the mutant will be killed under the weak mutation criterion. However, 6 and 0 are both even, so the decision starting on statement 5 will return `true` for both the mutated and non-mutated versions. That is, propagation is not satisfied, so test case `X = -6` will not kill the mutant under the strong mutation criterion. The propagation condition for this mutant is that the number be odd. Thus, to satisfy the strong mutation criterion, we require $(X < 0) \wedge (X \neq 0) \wedge odd(X)$, which can be simplified to $X$ must be an odd, negative integer.

### Testing Programs with Mutation

A test process gives a sequence of steps to follow to generate test cases. A single criterion may be used with many processes, and a test process may not even include a criterion. Many people find mutation less intuitive than other coverage criteria. The idea of "killing" a mutant is not as obvious as "reaching" a node, "traversing" a path, or "satisfying" a set of truth assignments. It is clear however, that the software is tested, and tested well, or the test cases do not kill mutants. This point can best be understood by examining a typical mutation analysis process.

Figure 9.2 shows how mutation testing can be applied. The tester submits the program under test to an automated system, which starts by creating mutants. Optionally, those mutants are then analyzed by a heuristic that detects and eliminates as many equivalent mutants as possible[2]. A set of test cases is then generated automatically and executed first against the original program, and then the mutants. If the output of a mutant program differs from the original (correct) output, the mutant is marked as being dead and is considered to have been *strongly killed* by that test case. Dead mutants are not executed against subsequent test cases. Test cases that do not strongly kill at least one mutant are considered

[2]Of course, since mutant detection is undecidable, a heuristic is the best option possible.

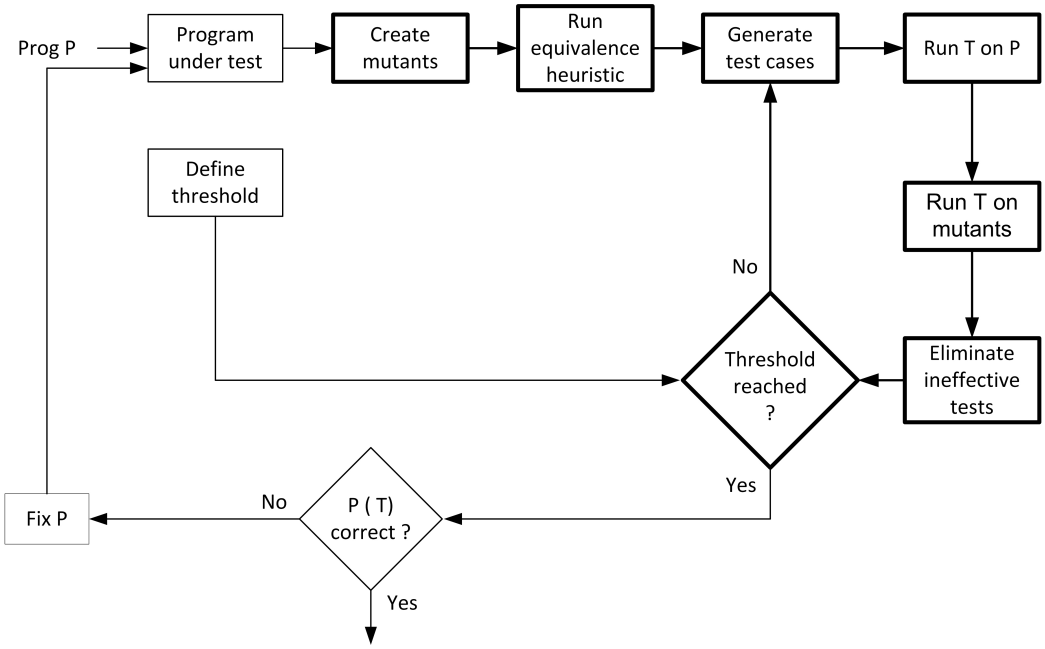


Figure 9.2: Mutation testing process.
**Bold boxes represent steps that are automated; other boxes represent manual steps.**

to be "ineffective" and eliminated, even though such test cases may weakly kill one or more mutants. This is because the requirement stated above requires the output (and not the internal state) to be different.

Once all test cases have been executed, coverage is computed as a mutation score. The mutation score is the ratio of dead mutants over the total number of non-equivalent mutants. If the mutation score reaches 1.00, that means all mutants have been detected. A test set that kills all the mutants is said to be *adequate* relative to the mutants.

A mutation score of 1.00 is usually impractical, so the tester defines a "threshold" value, which is a minimum acceptable mutation score. If the threshold has not been reached, then the process is repeated, each time generating test cases to target live mutants, until the threshold mutation score is reached. Up to this point, the process has been entirely automatic. To finish testing, the tester will examine expected output of the effective test cases, and fix the program if any faults are found. This leads to the fundamental premise of mutation testing: **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 that fault**.

### Designing Mutation Operators

Mutation operators must be chosen for each language and although they overlap quite a bit, some differences are particular to the language, often depending on the language features.

Researchers have designed mutation operators for many programming languages, including Fortran IV, COBOL, Fortran 77, C, C integration testing, Lisp, Ada, Java, and Java class relationships. Researchers have also designed mutation operators for the formal specification language SMV (discussed in Section 9.4.2), and for XML messages (discussed in Section 9.5.2).

As a field, we have learned a lot about designing mutation operators over the years. Detailed lists of mutation operators for various languages are provided in the literature, as referenced in the bibliographic notes for this chapter. Mutation operators are generally designed either to mimic typical programmer mistakes, or to encourage testers to follow common testing heuristics. Operators that change relational operators or variable references are examples of operators that mimic typical programmer mistakes. The *failOnZero()* operator used in Figure 9.1 is an example of the latter design; the tester is encouraged to follow the common testing heuristic of "causing each expression to become zero."

When first designing mutation operators for a new language, it is reasonable to be "inclusive," that is, include as many operators as possible. However, this often results in a large number of mutation operators, and an even larger number of mutants. Researchers have devoted a lot of effort to trying to find ways to use fewer mutants and mutation operators. The two most common ways to have fewer mutants are (1) to randomly sample from the total number of mutants, and (2) to use mutation operators that are particularly effective.

The term *selective mutation* has been used to describe the strategy of using only mutation operators that are particularly effective. Effectiveness has been evaluated as follows: If tests that are created specifically to kill mutants created by mutation operator $o_i$ also kill mutants created by mutation operator $o_j$ with very high probability, then mutation operator $o_i$ is more *effective* than $o_j$.

This notion can be extended to consider a collection of effective mutation operators as follows:

**Definition 9.7 Effective Mutation Operators:** *If tests that are created specifically to kill mutants created by a collection of mutation operators $O = \{o_1, o_2, ...\}$ also kill mutants created by all remaining mutation operators with very high probability, then $O$ defines an effective set of mutation operators.*

Researchers have concluded that a collection of mutation operators that insert unary operators and that modify unary and binary operators will be **effective**. The actual research was done with Fortran-77 (the Mothra system), but the results are adapted to Java in this chapter. Corresponding operators can be defined for other languages. The operators defined below are used throughout the remainder of this chapter as the defining set of program-level mutation operators.

1. ***ABS—Absolute Value Insertion***:

| Each arithmetic expression (and subexpression) is modified by the functions *abs()*, *negAbs()*, and *failOnZero()*. |
|---|

*abs()* returns the absolute value of the expression and *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. This operator is designed specifically to force the tester to cause each numeric expression to have the value 0, a negative value, and a positive value. For example, the statement `"x = 3 * a;"` is mutated to create the following statements:

```
x = 3 * abs (a);
x = 3 * - abs (a);
x = 3 * failOnZero (a);
x = abs (3 * a);
x = - abs (3 * a);
x = failOnZero (3 * a);
```

2. ***AOR—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, and *mod* computes the remainder when the left operand is divided by the right. For example, the statement `"x = a + b;"` is mutated to create the following seven statements:

```
x = a - b;
x = a * b;
x = a / b;
x = a ** b;
x = a;
x = b;
x = a % b;
```

3. ***ROR—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*. For example, the statement `"if (m > n)`" is mutated to create the following seven statements:

```
if (m >= n)
if (m < n)
if (m <= n)
if (m == n)
if (m != n)
if (false)
if (true)
```

4. ***COR—Conditional Operator Replacement***:

| Each occurrence of each logical operator (*and*–&&, *or*–‖, *and with no conditional evaluation*–&, *or with no conditional evaluation*–\|, and *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. *falseOp* always returns *false* and *trueOp* always returns *true*. For example, the statement `"if (a && b)"` is mutated to create the following eight statements:

```
if (a || b)
if (a & b)
if (a | b)
if (a ^ b)
if (false)
if (true)
if (a)
if (b)
```

5. ***SOR—Shift Operator Replacement***:

| Each occurrence of one of the shift operators <<, >>, and >>> is replaced by each of the other operators. In addition, each is replaced by the special mutation operator *leftOp*. |
|---|

*leftOp* returns the left operand unshifted. For example, the statement `"x = m << a;"` is mutated to create the following three statements:

```
x = m >> a;
x = m >>> a;
x = m;
```

6. ***LOR—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 (the right is ignored) and *rightOp* returns the right operand. For example, the statement `"x = m & n;"` is mutated to create the following four statements:

```
x = m | n;
x = m ^ n;
x = m;
x = n;
```

7. ***ASR—Assignment Operator Replacement***:

| Each occurrence of one of the assignment operators (=, +=, -=, *=, /=, %=, &=, \|=, ^=, <<=, >>=, >>>=) is replaced by each of the other operators. |
|---|

For example, the statement `"x += 3;"` is mutated to create the following ten statements:

```
x = 3;
x -= 3;
x *= 3;
x /= 3;
x %= 3;
x &= 3;
x |= 3;
x ^= 3;
x <<= 3;
x >>= 3;
x >>>= 3;
```

8. ***UOI—Unary Operator Insertion***:

| Each unary operator (*arithmetic* +, *arithmetic* -, *conditional* !, and *logical* ~) |
|---|

is inserted before each expression of the correct type.

For example, the statement `"x = 3 * a;"` is mutated to create the following four statements:

```
x = 3 * +a;
x = 3 * -a;
x = +3 * a;
x = -3 * a;
```

9. ***UOD—Unary Operator Deletion***:

| Each unary operator (*arithmetic* +, *arithmetic* -, *conditional* !, and *logical* ~) |
|---|

is deleted.

For example, the statement `"if !(a > -b)"` is mutated to create the following two statements:

```
if (a > -b)
if !(a > b)
```

Two other operators that are useful in examples are scalar variable replacement and the "bomb" operator. Scalar variable replacement results in a lot of mutants ($V^2$ if $V$ is the number of variables), and it turns out that it is not necessary given the above operators. It is included here as a convenience for examples. The bomb operator results in only one mutant per statement, but it is also not necessary given the above operators.

10. ***SVR—Scalar Variable Replacement***:

| Each variable reference is replaced by every other variable of the appropriate type that is declared in the current scope. |
|---|

For example, the statement `"x = a * b;"` is mutated to create the following six statements:

```
x = a * a;
a = a * b;
x = x * b;
x = a * x;
x = b * b;
b = a * b;
```

11. ***BSR—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. For example, the statement `"x = a * b;"` is mutated to create the following statement:

```
Bomb();
```

### Subsumption of Other Test Criteria (*Advanced Topic*)

Mutation is widely considered the strongest test criterion in terms of finding the most faults. It is also the most expensive. This section shows that mutation subsumes a number of other coverage criteria. The proofs are developed by showing that specific mutation operators impose requirements that are identical to a specific coverage criterion. For each specific requirement defined by a criterion, a single mutant is created that can be killed only by test cases that satisfy the requirement. Therefore, the coverage criterion is satisfied if and only if the mutants associated with the requirements for the criterion are killed. In this case, the mutation operators that ensure coverage of a criterion are said to *yield* the criterion. If a criterion is yielded by one or more mutation operators, then mutation testing subsumes the criterion. Although mutation operators vary by language and mutation analysis tool, this section uses common operators that are used in most implementations. It is also possible to design mutation operators to force mutation to subsume other testing criteria. Further details are given in the bibliographic notes.

This type of proof has one subtle problem. All previous coverage criteria impose only a **local** (reachability) requirement; for example, edge coverage requires each branch in the program to be executed. Mutation, on the other hand, imposes **global** (propagation) requirements in addition to local requirements. That is, mutation also requires that the mutant program produce incorrect output. For edge coverage, some specific mutants can be killed only if each branch is executed **and** the final output of the mutant is incorrect. On the one hand, this means that mutation imposes stronger requirements than the condition coverage

criteria. On the other hand, and somewhat perversely, this also means that sometimes a test set that satisfies a coverage criteria will not strongly kill all the associated mutants. Thus, mutation as defined earlier will not strictly subsume the condition coverage criteria.

This problem is solved by basing the subsumptions on *weak mutation*. In terms of subsuming other coverage criteria, weak mutation only imposes the local requirements. In weak mutation, mutants that are **not** equivalent at the infection stage but **are** equivalent at the propagation stage (that is, an incorrect state is masked or repaired) are left in the set of test cases, so that edge coverage is subsumed. It is precisely the fact that such test cases are removed that strong mutation does not subsume edge coverage.

Thus, this section shows that the coverage criteria are subsumed by weak mutation, not strong mutation.

Subsumption is shown for graph coverage criteria from Chapter 7 and logic coverage criteria from Chapter 8. Some mutation operators only make sense for program source statements whereas others can apply to arbitrary structures such as logical expressions. For example, one common mutation operator is to replace statements with "bombs" that immediately cause the program to terminate execution or raise an exception. This mutation can only be defined for program statements. Another common mutation operator is to replace relational operators ($<$, $>$, etc.) with other relational operators (the ROR operator). This kind of relational operator replacement can be applied to any logical expression, including guards in FSMs.

**Node coverage** requires each statement or basic block in the program to be executed. The mutation operator that replaces statements with "bombs" yields node coverage. To kill these mutants, we are required to design test cases that reach each basic block. Since this is exactly the requirement of node coverage, this operator yields node coverage and mutation subsumes node coverage.

**Edge coverage** requires each edge in the control flow graph to be executed. The ROR mutation operator replaces each predicate with both *true* and *false*. 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 operator forces each branch in the program to be executed, and thus it yields edge coverage and mutation subsumes edge coverage.

**Clause coverage** requires each clause to become both *true* and *false*. The ROR, COR, and LOR mutation operators will together replace each clause in each predicate with both *true* and *false*. To kill the *true* mutant, a test case must cause the clause (and also the full predicate) to be *false*, and to kill the *false* mutant, a test case must cause the clause (and also the full predicate) to be *true*. This is exactly the requirement for clause coverage. A simple way to illustrate this is with a modified form of a truth table.

Consider a predicate that has two clauses connected by an `AND`. Assume the predicate is $(a \wedge b)$, where $a$ and $b$ are arbitrary boolean-valued clauses. The partial truth table in Figure 9.3 shows $(a \wedge b)$ on the top line with the resulting value for each of the four combinations of values for $a$ and $b$. Below the line are four mutations that replace each of $a$ and $b$ with *true* and *false*. To kill the mutants, the tester must choose an input (one of the four truth assignments on top of the table) that causes a result that is different from that of the original

predicate. Consider mutant 1, $true \wedge b$. Mutant 1 has the same result as the original clause for three of the four truth assignments. Thus, to kill that mutant, the tester must use a test case input value that causes the truth assignment (F T), as shown in the box. Likewise, mutant 3, $a \wedge true$, can be killed only if the truth assignment (T F) is used. Thus, mutants 1 and 3 are killed if and only if clause coverage is satisfied, and the mutation operator yields clause coverage for this case. Note that mutants 2 and 4 are not needed to subsume clause coverage.

| | | (T T) | (T F) | (F T) | (F F) |
|---|---|---|---|---|---|
| | $a \wedge b$ | T | F | F | F |
| 1 | $true \wedge b$ | T | F | **T** | F |
| 2 | $false \wedge b$ | **F** | F | F | F |
| 3 | $a \wedge true$ | T | **T** | F | F |
| 4 | $a \wedge false$ | **F** | F | F | F |

Figure 9.3: Partial truth table for $(a \wedge b)$.

Although the proof technique of showing that mutation operators yield clause coverage on a case-by-case basis with the logical operators is straightforward and relatively easy to grasp, it is clumsy. More generally, assume a predicate $p$ and a clause $a$, and the clause coverage requirements to test $p(a)$, which says that $a$ must evaluate to both *true* and *false*. Consider the mutation $\Delta p(a \rightarrow true)$ (that is, the predicate where $a$ is replaced by *true*). The only way to satisfy the infection condition for this mutant (and thus kill it) is to find a test case that causes $a$ to take on the value of *false*. Likewise, the mutation $\Delta p(a \rightarrow false)$ can be killed only by a test case that causes $a$ to take on the value of *true*. Thus, in the general case, the mutation operator that replaces clauses with *true* and *false* yield clause coverage and is subsumed by mutation.

**Combinatorial coverage** requires that the clauses in a predicate evaluate to each possible combination of truth values. In the general case combinatorial coverage has $2^N$ requirements for a predicate with $N$ clauses. Since no single or combination of mutation operators produces $2^N$ mutants, it is easy to see that mutation cannot subsume COC.

**Active clause coverage** requires that each clause $c$ in a predicate $p$ evaluates to *true* and *false* **and** determines the value of $p$. The first version in Chapter 8, **General Active Clause Coverage** allows the values for other clauses in $p$ to have different values when $c$ is true and $c$ is false. It is simple to show that mutation subsumes General Active Clause Coverage; in fact, we already have.

To kill the mutant $\Delta p(a \rightarrow true)$, we must satisfy the infection condition by **causing** $p(a \rightarrow true)$ **to have a different value from** $p(a)$, that is, $a$ must determine $p$. Likewise, to kill $\Delta p(a \rightarrow false)$, $p(a \rightarrow false)$ must have a different result from $p(a)$, that is, $a$ must determine $p$. Since this is exactly the requirement of GACC, this operator yields node coverage and mutation subsumes general active clause coverage. Note that this is only true if the incorrect value in the mutated program propagates to the end of the expression, which is one interpretation of weak mutation.

Neither **Correlated Active Clause Coverage** nor **Restricted Active Clause Coverage** are subsumed by mutation operators. The reason is that both CACC and RACC require pairs of tests to have certain properties. In the case of CACC, the property is that the predicate outcome be different on the two tests associated with a particular clause. In the case of RACC, the property is that the minor clauses have exactly the same values on the two tests associated with a particular clause. Since each mutant is killed (or not) by a single test case, (as opposed to a pair of test cases), mutation analysis, at least as traditionally defined, cannot subsume criteria that impose relationships between pairs of test cases.

Researchers have not determined whether mutation subsumes the inactive clause coverage criteria.

**All-defs data flow coverage** requires that each definition of a variable reach at least one use. That is, for each definition of a variable $X$ on node $n$, there must be a definition-clear subpath for $X$ from $n$ to a node or an edge with a use of $n$. The argument for subsumption is a little complicated for All-defs, and unlike the other arguments, all-defs requires that strong mutation be used.

A common mutation operator is to delete statements with the goal of forcing each statement in the program to make an impact on the output[3]. To show subsumption of All-defs, we restrict our attention to statements that contain variable definitions. Assume that the statement $s_i$ contains a definition of a variable $x$, and $m_i$ is the mutant that deletes $s_i$ ($\Delta s_i \rightarrow null$). To kill $m_i$ under strong mutation, a test case $t$ must (1) cause the mutated statement to be reached (reachability), (2) cause the execution state of the program after execution of $s_i$ to be incorrect (infection), and (3) cause the final output of the program to be incorrect (propagation). Any test case that reaches $s_i$ will cause an incorrect execution state, because the mutated version of $s_i$ will not assign a value to $x$. For the final output of the mutant to be incorrect, two cases are possible. First, if $x$ is an output variable, $t$ must have caused an execution of a subpath from the deleted definition of $x$ to the output without an intervening definition (def-clear). Since the output is considered a use, this satisfies the criterion. Second, if $x$ is not an output variable, then not defining $x$ at $s_i$ must result in an incorrect output state. This is possible only if $x$ is used at some later point during execution without being redefined. Thus, $t$ satisfies the all-defs criterion for the definition of $x$ at $s_i$, and the mutation operator yields all-defs, ensuring that mutation subsumes all-defs.

It is possible to design a mutation operator specifically to subsume all-uses, but such an operator has never been published or used in any tool.

---

**Exercises, Section 9.2.**

1. Provide reachability conditions, infection conditions, propagation conditions, and test case values to kill mutants 2, 4, 5, and 6 in Figure 9.1.

[3]This goal is in some sense equivalent to the goal of forcing each clause in each predicate to make a difference.

2. Answer questions (a) through (d) for the mutants in the two methods, `findVal()` and `sum()`.

   (a) If possible, find test inputs that do **not** reach the mutant.
   (b) If possible, find test inputs that satisfy reachability but **not infection** for the mutant.
   (c) If possible, find test inputs that satisfy infection, but **not propagation** for the mutant.
   (d) If possible, find test inputs that kill the mutants.

```
/**
 * Find last index of element
 *
 * @param numbers array to search
 * @param val value to look for
 * @return last index of val in numbers; -1 if absent
 * @throws NullPointerException if numbers is null
 */
1. public static int findVal(int numbers[], int val)
2. {
3.    int findVal = -1;
4.
5.    for (int i=0; i<numbers.length; i++)
5'.// for (int i=(0+1); i<numbers.length; i++)
6.       if (numbers [i] == val)
7.          findVal = i;
8.    return (findVal);
9. }
```

```
/**
 * Sum values in an array
 *
 * @param x array to sum
 *
 * @return sum of values in x
 * @throws NullPointerException if x is null
 */
1. public static int sum(int[] x)
2. {
3.    int s = 0;
4.    for (int i=0; i < x.length; i++) }
5.    {
6.       s = s + x[i];
6'.   // s = s - x[i]; //AOR
7.    }
8.    return s;
9.  }
```

3. Refer to the `patternIndex()` method in the `PatternIndex` program in Chapter 7. Consider Mutant A and Mutant B given below. Implementations are available on the book website in files `PatternIndexA.java` and `PatternIndexB.java`.

```
while (isPat == false && isub + patternLen - 1 < subjectLen) // Original
while (isPat == false && isub + patternLen - 0 < subjectLen) // Mutant A

isPat = false;  // Original  (Inside the loops, not the declaration)
isPat = true;   // Mutant B
```

   Answer the following questions for each mutant.

   (a) If possible, design test inputs that do **not reach** the mutants.
   (b) If possible, design test inputs that satisfy reachability but **not infection** for the mutants.
   (c) If possible, design test inputs that satisfy reachability and infection, but **not propagation** for the mutants.
   (d) If possible, design test inputs that strongly kill the mutants.

4. Why does it make sense to remove ineffective test cases?

5. Define 12 mutants for the following method *cal()* using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for *cal()* were created?

```
public static int cal (int month1, int day1, int month2, int day2, int year)
{
//*********************************************************
// Calculate the number of Days between the two given days in
// the same year.
// preconditions : day1 and day2 must be in same year
//               1 <= month1, month2 <= 12
//               1 <= day1, day2 <= 31
//               month1 <= month2
//               The range for year: 1 ... 10000
//*********************************************************
  int numDays;

   if (month2 == month1) // in the same month
      numDays  = day2 - day1;
   else
   {
      // Skip month 0.
      int daysIn[] = {0, 31, 0, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
      // Are we in a leap year?
      int m4 = year % 4;
      int m100 = year % 100;
      int m400 = year % 400;
      if ((m4 != 0) || ((m100 ==0) && (m400 != 0)))
         daysIn[2] = 28;
      else
         daysIn[2] = 29;

      // start with days in the two months
      numDays = day2 + (daysIn[month1] - day1);

      // add the days in the intervening months
      for (int i = month1 + 1; i <= month2-1; i++)
         numDays = daysIn[i] + numDays;
   }
   return (numDays);
}
```

6. Define 12 mutants for the following method *power()* using the effective mutation operators given previously. Try to use each mutation operator at least once. Approximately how many mutants do you think there would be if all mutants for *power()* were created?

```
 public static int power (int left, int right)
 {
 //************************************
 // Raises left to the power of right
 // precondition : right >= 0
 // postcondition: Returns left**right
 //************************************
    int rslt;
    rslt = left;
    if (right == 0)
    {
       rslt = 1;
    }
    else
    {
       for (int i = 2; i <= right; i++)
           rslt = rslt * left;
    }
    return (rslt);
 }
```

7. The fundamental premise was stated as: "*In practice, if the software contains a fault, there*

*will usually be a set of mutants that can be killed only by a test case that also detects that fault.*"

(a) Give a brief argument **in support of** the fundamental mutation premise.
(b) Give a brief argument **against** the fundamental mutation premise.

8. Try to design mutation operators that subsume Combinatorial Coverage. Why wouldn't we want such an operator?

9. Look online for the tool Jester (`http://jester.sourceforge.net/`), which is based on JUnit. Based on your reading, evaluate Jester as a mutation-testing tool.

10. Download and install the Java mutation tool *muJava* from the book website Enclose the method *cal()* from question 5 inside a class, and use muJava to test *cal()*. Use all the operators. Design tests to kill all non-equivalent mutants. Note that a test case is a method call to *cal()*.

(a) How many mutants are there?
(b) How many test cases do you need to kill the non-equivalent mutants?
(c) What mutation score were you able to achieve before analyzing for equivalent mutants?
(d) How many equivalent mutants are there?

---

## 9.3 Integration and Object-Oriented Testing

This book defined the term *integration testing* in Chapter 2 as testing connections among separate program units. In Java, that involves testing the way classes, packages, and components are connected. This section uses the general term *component*. This is also where features that are unique to object-oriented programming languages are tested, specifically, inheritance, polymorphism, and dynamic binding.

### 9.3.1 BNF Integration Testing

As far as we know, BNF testing has not been used at the integration level.

### 9.3.2 Integration Mutation

This section first discusses how mutation can be used for testing at the integration level without regard to object-oriented relationships, then how mutation can be used to test for problems involving inheritance, polymorphism, and dynamic binding.

Faults that can occur in the integration between two components usually depend on a mismatch of assumptions. For example, Chapter 1 discussed the Mars lander of September 1999, which crashed because a component sent a value in English units (miles) and the

recipient component assumed the value was in kilometers. Whether such a flaw should be fixed by changing the caller, the callee, or both depends on the design specification of the program and possibly pragmatic issues such as which is easier to change.

*Integration mutation* (also called *interface mutation*) works by mutating the connections between components. Most mutants are around method calls, and both the calling (caller) and called (callee) method must be considered. Interface 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 as well as variables from a higher scope (class level, package, public, etc.).
- Change a called method by modifying statements that return from the method.

1. ***IPVR—Integration Parameter Variable Replacement***:

| 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 (the compiler should catch them). In OO languages, this operator replaces primitive type variables as well as objects.

2. ***IUOI—Integration Unary Operator Insertion***:

| Each expression in a method call is modified by inserting all possible unary operators in front of and behind it. |
|---|

The unary operators vary by language and type. Java includes `++` and `--` as both prefix and postfix operators for numeric types.

3. ***IPEX—Integration Parameter Exchange***:

| Each parameter in a method call is exchanged with each parameter of compatible type in that method call. |
|---|

For example, if a method call is `max (a, b)`, a mutated method call of `max (b, a)` is created.

4. ***IMCD—Integration Method Call Deletion***:

| 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 the method returns an object, the method call should be replaced by a call to **new()** on the appropriate class.

5. ***IREM—Integration Return Expression Modification***:

| Each expression in each return statement in a method is modified by applying the UOI and AOR operators from Section 9.2.2. |
|---|

### Object-Oriented Mutation Operators

Chapter 2 defined intra-method, inter-method, intra-class, and inter-class testing. The five integration mutation operators can be used at the inter-method level (between methods in the same class) and at the inter-class level (between methods in different classes). When testing at the inter-class level, testers also have to worry about faults in the use of inheritance and polymorphism. These are powerful language features that can solve difficult programming problems, but also introduce difficult testing problems.

Languages that include features for inheritance and polymorphism often also include features for information hiding and overloading. Thus, mutation operators to test those features are usually included with the OO operators, even though these are not usually considered to be essential to calling a language "object-oriented."

To understand how mutation testing is applied to such features, we need to examine the language features in depth. This is done in terms of Java; other OO languages tend to be similar but with some subtle differences.

*Encapsulation* is an abstraction mechanism to enforce information hiding, a design technique that frees clients of an abstraction from unnecessary dependence on design decisions in the implementation of the abstraction. Encapsulation allows objects to restrict access to their member variables and methods by other objects. Java supports four distinct access levels for member variables and methods: private, protected, public, and default (also called package). Many programmers do not understand these access levels well, and often do not consider them during design, so they are a rich source of faults. Table 9.1 summarizes these access levels. A *private* member is available only to the class in which it is defined. If access is not specified, the access level defaults to *package*, which allows access to classes in the same package, but **not** subclasses in other packages. A *protected* member is available to the class itself, subclasses, and classes in the same package. A *public* member is available to any class in any inheritance hierarchy or package (the world).

Table 9.1: Java's access levels.

| Specifier | Same class | Different class/ same package | Different package subclass | Different package non-subclass |
|---|---|---|---|---|
| private | Y | n | n | n |
| package | Y | Y | n | n |
| protected | Y | Y | Y | n |
| public | Y | Y | Y | Y |

Java does not support multiple class inheritance, so every class has only one immediate parent. A subclass inherits variables and methods from its parent and all of its ancestors, and can use them as defined, or override the methods or hide the variables. Subclasses can also explicitly use their parent's variables and methods using the keyword "super" (`super.methodname();`). Java's inheritance allows method overriding, variable hiding, and class constructors.

*Method overriding* allows a method in a subclass to have the same name, arguments and result type as a method in its parent. Overriding allows subclasses to redefine inherited methods. The child class method has the same signature, but a different implementation.

*Variable hiding* is achieved by defining a variable in a child class that has the same name and type of an inherited variable. This has the effect of hiding the inherited variable from the child class. This is a powerful feature, but it is also a potential source of errors.

*Class constructors* are not inherited in the same way other methods are. To use a constructor of the parent, we must explicitly call it using the *super* keyword. The call must be the first statement in the derived class constructor and the parameter list must match the parameters in the argument list of the parent constructor.

Java supports two versions of polymorphism, attributes and methods, both of which use dynamic binding. Each object has a *declared type* (the type in the declaration statement, that is, "*Parent P;*") and an *actual type* (the type in the instantiation statement, that is, "*P = new Child();*," or the assignment statement, "*P = Pold;*"). The actual type can be the declared type or any type that is descended from the declared type.

A *polymorphic attribute* is an object reference that can take on various types. At any location in the program, the type of the object reference can be different in different executions. A *polymorphic method* can accept parameters of different types by having a parameter that is declared of type *Object*. Polymorphic methods are used to implement *type abstraction* (templates in C++ and generics in Ada).

*Overloading* is the use of the same name for different constructors or methods in the same class. They must have different *signatures*, or lists of arguments. Overloading is easily confused with overriding because the two mechanisms have similar names and semantics. Overloading occurs with two methods in the same class, whereas overriding occurs between a class and one of its descendants.

In Java, member variables and methods can be associated with the class rather than with individual objects. Members associated with a class are called *class* or *static variables* and *methods*. The Java runtime system creates a single copy of a static variable the first time it encounters the class in which the variable is defined. All instances of that class share the same copy of the static variable. Static methods can operate only on static variables; they cannot access instance variables defined in the class. Unfortunately the terminology varies; we say *instance variables* are declared at the class level and are available to objects, *class variables* are declared with `static`, and *local variables* are declared within methods.

Mutation operators can be defined for all of these language features. The purpose of mutating them is to make sure that the programmer is using them correctly. One reason to be particularly concerned about the use of OO language features is because many programmers today have learned them "on the job," without having the opportunity to study the theoretical rules about how to use them appropriately.

Following are 20 mutation operators for information hiding language features, inheritance, polymorphism and dynamic binding, method overloading, and classes.

**Group 1: Encapsulation mutation operators**

1. ***AMC—Access Modifier Change***:

| |
|---|
| The access level for each instance variable and method is changed to other access levels. |

The AMC operator helps testers generate tests to ensure that accessibility is correct. These mutants can be killed only if the new access level denies access to another class or allows access that causes a name conflict.

**Group 2: Inheritance mutation operators**

2. ***IHI—Hiding Variable Insertion***:

| |
|---|
| A declaration is added to hide the declaration of each variable declared in an ancestor. |

These mutants can be killed only by test cases that can show that the reference to the overriding variable is incorrect.

3. ***IHD—Hiding Variable Deletion***:

| |
|---|
| Each declaration of an overriding (hiding), variable is deleted. |

This causes references to that variable to access the variable defined in the parent (or ancestor), which is a common programming mistake.

4. ***IOD—Overriding Method Deletion***:

| |
|---|
| Each entire declaration of an overriding method is deleted. |

References to the method will then use the parent's version. This ensures that the method invocation is to the intended method.

5. ***IOP—Overridden Method Calling Position Change***:

| |
|---|
| Each call to an overridden method is moved to the first and last statements of the method and up and down one statement. |

Overriding methods in child classes often call the original method in the parent class, for example to modify a variable that is private to the parent. A common mistake is to call the parent's version at the wrong time, which can cause incorrect state behavior.

6. ***IOR—Overridden Method Rename***:

| |
|---|
| Renames the parent's versions of methods that are overridden in a subclass so that the overriding does not affect the parent's method. |

The IOR operator is designed to check whether an overriding method causes problems with other methods. Consider a method *m()* that calls another method *f()*, both in a class *List*. Further, assume that *m()* is inherited without change in a child class *Stack*, but *f()* is overridden in *Stack*. When *m()* is called on an object of type *Stack*, it calls *Stack*'s version of *f()* instead of *List*'s version. In this case, *Stack*'s version of *f()* may interact with the parent's version that has unintended consequences.

7. ***ISI—super Keyword Insertion***:

| |
|---|
| Inserts the *super* keyword before overriding variables or methods (if the name is also defined in an ancestor class). |

After the change, references will be to an ancestor's version. The ISI operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

8. ***ISD—super Keyword Deletion***:

| Delete each occurrence of the *super* keyword. |
|---|

After the change, the reference will be to the local version instead of the ancestor's version. The ISD operator is designed to ensure that hiding/hidden variables and overriding/overridden methods are used appropriately.

9. ***IPC—Explicit Parent's Constructor Deletion***:

| Each call to a *super* constructor is deleted. |
|---|

The parent's (or ancestor's) default constructor will be used. To kill these mutants, it is necessary to find a test case for which the parent's default constructor creates an initial state that is incorrect.

**Group 3: Polymorphism mutation operators**

10. ***PNC—new Method Call With Child Class Type***:

| The actual type of a new object is changed in the `new()` statement. |
|---|

This causes the object reference to refer to an object of a type that is different from the original actual type. The new actual type must be in the same "type family" (a descendant) of the original actual type.

11. ***PMD—Member Variable Declaration with Parent Class Type***:

| The declared type of each new object is changed in the declaration. |
|---|

The new declared type must be an ancestor of the original type. The instantiation will still be valid (it will still be a descendant of the new declared type). To kill these mutants, a test case must cause the behavior of the object to be incorrect with the new declared type.

12. ***PPD—Parameter Variable Declaration with Child Class Type***:

| The declared type of each parameter object is changed in the declaration. |
|---|

This is the same as PMD except on parameters.

13. ***PCI—Type cast operator insertion***:

| The actual type of an object reference is changed to the parent or to the child of the original declared type. |
|---|

The mutant will have different behavior when the object to be cast has hiding variables or overriding methods.

14. ***PCD—Type cast operator deletion***:

| The PCD operator deletes type casting operators. |
|---|

This operator is the inverse of PCI.

15. ***PCC—Cast type change***:

| The PCC operator changes the type to which an object reference is being cast. |
|---|

The new type must be in the type hierarchy of the declared type (that is, it must be a valid cast).

16. ***PRV—Reference Assignment with Other Compatible Type***:

| The right side objects of assignment statements are changed to refer to objects of a compatible type. |
|---|

For example, if an `Integer` is assigned to a reference of type `Object`, the assignment may be changed to that of a `String`. Since both `Integer`s and `String`s descend from `Object`, both can be assigned interchangeably.

17. ***OMR—Overloading Method Contents Replace***:

| For each pair of methods that have the same name, the bodies are interchanged. |
|---|

This ensures that overloaded methods are invoked appropriately.

18. ***OMD—Overloading Method Deletion***:

| Each overloaded method declaration is deleted, one at a time. |
|---|

The OMD operator ensures coverage of overloaded methods; that is, all the overloaded methods must be invoked at least once. If the mutant still works correctly without the deleted method, there may be an error in invoking one of the overloading methods; the incorrect method may be invoked, or an incorrect parameter type conversion has occurred.

19. ***OAC—Arguments of Overloading Method Call Change***:

| The order of the arguments in method invocations is changed to be the same as that of another overloading method, if one exists. |
|---|

This causes a different method to be called, thus checking for a common fault in the use of overloading.

**Group 4: Java-specific mutation operators**

20. ***JTI–this Keyword Insertion***:

| The keyword *this* is inserted whenever possible. |
|---|

Within a method body, uses of the keyword *this* refers to the current object if the member variable is hidden by a local variable or method parameter that has the same name. JTI replaces occurrences of "$X$" with "*this.*$X$." JTI mutants are killed when using the local version instead of the current object changes the behavior of the software.

21. ***JTD—this Keyword Deletion***:

| Each occurrence of the keyword *this* is deleted. |
|---|

The JTD operator checks if the member variables are used correctly by replacing occurrences of "*this.*$X$" with "$X$."

22. ***JSI–static modifier insertion***:

| The *static* modifier is added to instance variables. |
|---|

This operator ensures that variables that are declared as non-static really need to be non-static.

23. ***JSD–static modifier deletion***:

| Each instance of the *static* modifier is removed |
|---|

This operator ensures that variables that are declared as static really need to be static.

24. ***JID—Member Variable Initialization Deletion***:

| Remove initialization of each member variable. |
|---|

Instance variables can be initialized in the variable declaration and in constructors for the class. The JID operator removes the initializations so that member variables are initialized to the default values.

25. ***JDC—Java-supported Default Constructor Deletion***:

| Delete each declaration of a default constructor |
|---|

This ensures that default constructors are implemented correctly.

## 9.4 Specification-based Grammars

The general term "specification-based" is applied to languages that describe software in abstract terms. This includes formal specification languages such as Z, SMV, OCL, etc., and informal specification languages and design notations such as statecharts, FSMs, and other UML diagram notations. Design notations are also referred to as "model-based." Thus, the line between specification-based and model-based is blurry. Such languages are becoming more widely used, partly because of increased emphasis on software quality and partly because of the widespread use of the UML.

### 9.4.1 BNF Grammars

To our knowledge, terminal symbol coverage and production coverage have been applied to only one type of specification language: algebraic specifications. The idea is to treat an equation in an algebraic specification as a production rule in a grammar, and then derive strings of method calls to cover the equations. As algebraic specifications are not widely used, this book does not discuss this topic.

### 9.4.2 Specification-based Mutation

Mutation testing can also be a valuable method at the specification level. In fact, for certain types of specifications, mutation analysis is actually easier. We address specifications expressed as finite state machines in this section.

A finite state machine is essentially a graph $G$, as defined in Chapter 7, with a set of states (nodes), a set of initial states (initial nodes), and a transition relation (the set of

edges). When finite state machines are used, sometimes the edges and nodes are explicitly identified, as in the typical bubble and arrow diagram. However, sometimes the finite state machine is more compactly described in the following way:

1. States are implicitly defined by declaring variables with limited ranges. The state space is then the Cartesian product of the ranges of the variables.

2. Initial states are defined by limiting the ranges of some or all of the variables.

3. Transitions are defined by rules that characterize the source and target of each transition.

The following example clarifies these ideas in the language SMV. We describe a machine with a simple syntax, and show the same machine with explicit enumerations of the states and transitions. Although this example is too small to show this point, the syntax version in SMV is typically *much* smaller than the graph version. In fact, since state space growth is combinatorial, it is quite easy to define finite state machines where the explicit version is far too long to write, even though the machine itself can be analyzed efficiently. Below is an example in the SMV language.

```
MODULE main
#define false 0
#define true 1

VAR
   x, y : boolean;

ASSIGN
   init (x) := false;
   init (y) := false;

   next (x) := case
      !x & y : true;
      !y     : true;
      x      : false;
      true   : x;
   esac;

   next (y) := case
      x & !y : false;
      x & y  : y;
      !x & y : false;
      true   : true;
   esac;
```

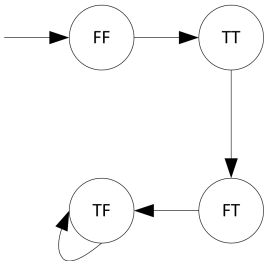


Figure 9.4: Finite state machine for SMV specification.

Two variables appear, each of which can have only two values (boolean), so the state space is of size $2 * 2 = 4$. One initial state is defined in the two `init` statements under `ASSIGN`. The transition diagram is shown in Figure 9.4. Transition diagrams for SMV can be derived by mechanically following the specifications. Take a given state and decide what the next value for each variable is. For example, assume the above specification is in the state (*true*, *true*). The next value for $x$ will be determined by the "`x : false`" statement. $x$ is *true*, so its next value will be $false$. Likewise, $x$ & $y$ is true, so the next value of $y$ will be its current value, or *true*. Thus, the state following (*true*, *true*) is ($false$, *true*). If multiple conditions in a `case` statement are true, the first one that is true is chosen. SMV has no "fall-through" semantics, such as in languages like C or Java.

Our context has two particularly important aspects of such a structure.

1. Finite state descriptions can capture system behavior at a very high level–suitable for communicating with the end user. Finite state machines are incredibly useful for the hardest part of testing, namely system testing.

2. The verification community has built powerful analysis tools for finite state machines. These tools are highly automated. Further, these tools produce explicit evidence, in the form of witnesses or counterexamples, for properties that do not hold in the finite state machine. These counterexamples can be interpreted as test cases. Thus, it is easier to automate test case generation from finite state machines than from program source.

### Mutations and Test Cases

Mutating the syntax of state machine descriptions is very much like mutating program source. Mutation operators must be defined, and then they are applied to the description. One example is the *Constant Replacement* operator, which replaces each constant with other constants. Given the phrase `!x & y : false` in the `next` statement for `y`, replace it with `!x & y : true`. The finite state machine for this mutant is shown in Figure 9.5. The new transition is drawn as an extra thick arrow and the replaced transition is shown as a crossed-out dotted arrow.

Generating a test case to kill this mutant is a little different from program-based mutation. We need a sequence of states that is allowed by the transition relation of the original state

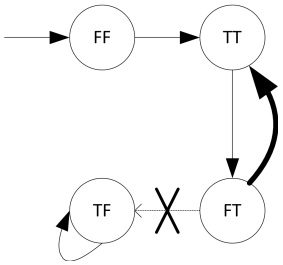


Figure 9.5: Mutated finite state machine for SMV specification.

machine, but not by the mutated state machine. Such a sequence is precisely a test case that kills the mutant.

Jia and Harman [34, 29] discovered that higher order mutants (HOMs), where more than one change is made at the same time, can be very helpful. They are primarily useful when the two changes interact, but do not cancel each other out.

Finding a test to kill a mutant of a finite state machine expressed in SMV can be automated using a *model checker*. A model checker takes two inputs. The first is a finite state machine, described in a formal language such as SMV. The second is a statement of some property, expressed in a *temporal logic*. We will not fully explain temporal logic here, other than to say that such a logic can be used to express properties that are true "now," and also properties that will (or might) be true in the future. The following is a simple temporal logic statement:

> The original expression, `!x & y : false` in this case, is **always** the same as the mutated expression, `x | y : true`.

For the given example, this statement is false with respect to a sequence of states allowed by the original machine if and only if that sequence of states is rejected by the mutant machine. In other words, such a sequence in question is a test case that kills the mutant. If we add the following SMV statement to the above machine:

`SPEC AG (!x & y)` $\longrightarrow$ AX `(y = true)`

The model checker will obligingly produce the desired test sequence:

```
/* state 1 */ { x  = 0, y  = 0 }
/* state 2 */ { x  = 1, y  = 1 }
/* state 3 */ { x  = 0, y  = 1 }
/* state 4 */ { x  = 1, y  = 0 }
```

Some mutated state machines are equivalent to the original machine. The model checker is exceptionally well adapted to deal with this. The key theoretical reason is that the model checker has a finite domain to work in, and hence the equivalent mutant problem is decidable (unlike with program code). In other words, if the model checker does not produce a counterexample, we *know* that the mutant is equivalent.

**Exercises, Section 9.4.**

1. (Challenging!) Find or write a small SMV specification and a corresponding Java implementation. Restate the program logic in `SPEC` assertions. Mutate the assertions systematically, and collect the traces from (nonequivalent) mutants. Use these traces to test the implementation.

## 9.5 Input Space Grammars

One common use of grammars is to define the syntax of the inputs to a program, method, or software component formally. This section explains how to apply the criteria of this chapter to grammars that define the input space of a piece of software.

### 9.5.1 BNF Grammars

Section 9.1.1 of this chapter presented criteria on BNF grammars. One common use of a grammar is to define a precise syntax for the input of a program or method.

Consider a program that processes a sequence of deposits and debits, where each deposit is of the form `deposit` *account amount* and each debit is of the form `debit` *account amount*. The input structure of this program can be described with the regular expression:

(`deposit` *account amount* | `debit` *account amount*)*

This regular expression describes any sequence of deposits and debits. (The example in Section 9.1.1 is actually an abstract version of this example.)

The regular expression input description is still fairly abstract, in that it does not say anything about what an *account* or an *amount* looks like. We will refine those details later. One input that can be derived from this grammar is:

```
deposit 739 $12.35
deposit 644 $12.35
debit 739 $19.22
```

It is easy to build a graph that captures the effect of regular expressions. Formally, these graphs are finite automata, either deterministic or nondeterministic. In either case, one can apply the coverage criteria from Chapter 7 directly.

One possible graph for the above structure is shown in Figure 9.6. It contains one state (`Ready`) and two transitions that represent the two possible inputs. The input test example given above satisfies both the all nodes and all edges criteria for this graph.

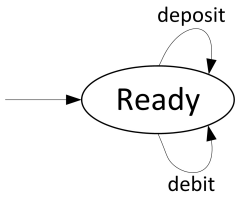


Figure 9.6: Finite state machine for bank example.

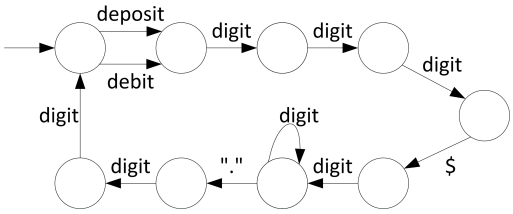


Figure 9.7: Finite state machine for bank example grammar.

Although regular expressions suffice for some programs, others require grammars. As grammars are more expressive than regular expressions we do not need to use both. The prior example specified in grammar form, with all of the details for *account* and *amount*, is:

```
bank    ::= action*
action  ::= dep | deb
dep     ::= "deposit" account amount
deb     ::= "debit" account amount
account ::= digit^3
amount  ::= "$" digit+ "." digit^2
digit   ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

The graph for even this simple example is substantially larger once all details have been included. It is shown in Figure 9.7.

A full derivation of the test case above begins as follows:

```
stream → action^*
       → action action^*
       → dep action^*
       → deposit account amount action^*
       → deposit digit^3 amount action^*
       → deposit digit digit^2 amount action^*
       → deposit 7 digit^2 amount action^*
       → deposit 7 digit digit amount action^*
       → deposit 73 digit amount action^*
       → deposit 739 amount action^*
```

```
→ deposit 739 $ digit^+ . digit^2 action^*
→ deposit 739 $ digit^2 .  digit^2 action^*
→ deposit 739 $ digit digit .  digit^2 action^*
→ deposit 739 $1 digit .  digit^2 action^*
→ deposit 739 $12.  digit^2 action^*
→ deposit 739 $12.  digit digit action^*
→ deposit 739 $12.3 digit action^*
→ deposit 739 $12.35 action^*
⋮
```

Deriving tests from this grammar proceeds by systematically replacing the next nonterminal (`action`) with one of its productions. The exercises below ask for complete tests to satisfy Terminal Symbol Coverage and Production Coverage.

Of course, it often happens that an informal description of the input syntax is available, but not a formal grammar. This means that the test engineer is left with the engineering task of formally describing the input syntax. This process is **extremely** valuable, and will often expose ambiguities and omissions in the requirements and software. Thus, this step should be carried out early in development, definitely before implementation and preferably before design. Once defined, it is sometimes helpful to use the grammar directly in the program for execution-time input validation.

### XML Example

A language for describing inputs that is widely used is the *eXtensible Markup Language (XML)*. The most common use of XML is in web applications and web services, but XML's structure is generic enough to be useful in many contexts. XML is a language for describing, encoding and transmitting data. All XML "messages" (also sometimes called "documents") are in plain text and use a syntax similar to HTML. XML comes with a built-in language for describing the input messages in the form of a grammar, called *schemas*.

Like HTML, XML uses *tags*, which are textual descriptions of data enclosed in angle brackets ('<' and '>'). All XML messages must be *well-formed*, that is, have a single document element with other elements properly nested under it, and every tag must have a corresponding closing tag. A simple example XML message for books is shown in Figure 9.8. This example is used to illustrate the use of BNF testing on software that uses XML messages. The example lists two books. The tag names ("books," "book," "ISBN," etc.) should be self descriptive and the XML message forms an overall hierarchy.

XML documents can be constrained by grammar definitions written in *XML Schemas*. Figure 9.9 shows a schema for books. The schema says that a *books* XML message can contain an unbounded number of *book* tags. The *book* tags contain six pieces of information. Three, *title*, *author*, and *publisher*, are simple strings. One, *price*, is of type decimal (numeric), has two digits after the decimal point and the lowest value is 0. Two data elements, *ISBN* and *year*, are types that are defined later in the schema. The type *yearType* is an integer with

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Sample XML file for books-->
<books xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="C:\Books\books.xsd">
 <book>
    <ISBN>0471043281</ISBN>
    <title>The Art of Software Testing</title>
    <author>Glen Myers</author>
    <publisher>Wiley</publisher>
    <price>50.00</price>
    <year>1979</year>
 </book>
 <book>
    <ISBN>0442206720</ISBN>
    <title>Software Testing Techniques</title>
    <author>Boris Beizer</author>
    <publisher>Van Nostrand Reinhold, Inc</publisher>
    <price>75.00</price>
    <year>1990</year>
 </book>
</books>
```

Figure 9.8: Simple XML message for books.

four digits, and "isbnType" can have up to 10 numeric characters. Each book must have a *title*, *author*, *publisher*, *price*, and *year*, and *ISBN* is optional.

Given an XML schema, the criteria defined in Section 9.1.1 can be used to derive XML messages that serve as test inputs. Following the production coverage criteria would result in two XML messages for this simple schema, one that includes an ISBN and one that does not.

### 9.5.2 Mutating Input Grammars

It is quite common to require a program to reject malformed inputs, and this property should definitely be tested as a form of stress testing. It is the kind of thing that slips past the attention of programmers who are focused on happy paths, that is, making a program do what it is supposed to do.

Do invalid inputs really matter? From the perspective of program correctness, invalid inputs are simply those outside the precondition of a specified function. Formally speaking, a software implementation of that function can exhibit any behavior on inputs that do not satisfy the precondition. This includes failure to terminate, runtime exceptions, and "bus error, core dumps."

However, the correctness of the intended functionality is only part of the story. From a practical perspective, invalid inputs sometimes matter a great deal because they hold the key to unintended functionality. For example, unhandled invalid inputs often represent

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified">
 <xs:element name="books">
  <xs:annotation>
   <xs:documentation>XML Schema for Books</xs:documentation>
  </xs:annotation>
  <xs:complexType>
   <xs:sequence>
    <xs:element name="book" maxOccurs="unbounded">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
       <xs:element name="title" type="xs:string"/>
       <xs:element name="author" type="xs:string"/>
       <xs:element name="publisher" type="xs:string"/>
       <xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
       <xs:element name="year" type="yearType"/>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>

 <xs:simpleType name="yearType">
  <xs:restriction base="xs:int">
   <xs:totalDigits value="4"/>
  </xs:restriction>
 </xs:simpleType>
 <xs:simpleType name="isbnType">
  <xs:restriction base="xs:string">
   <xs:pattern value="[0-9]{10}"/>
  </xs:restriction>
 </xs:simpleType>
</xs:schema>
```

Figure 9.9: XML schema for books.

security vulnerabilities, allowing a malicious party to break the software. Invalid inputs often cause the software to behave in surprising ways, which malicious parties can use to their advantage. This is how the classic "buffer overflow attack" works. The key step in a buffer overflow attack is to provide an input that is too long to fit into the available buffer. Similarly, a key step in certain web browser attacks is to provide a string input that contains malicious HTML, Javascript, or SQL. Software should behave "reasonably" with invalid inputs. "Reasonable" behavior may not always be defined, but the test engineer is obliged to consider it anyway.

To support security as well as to evaluate the software's behavior, it is useful to produce test cases that contain invalid inputs. A common way to do this is to mutate a grammar. When mutating grammars, the mutants are the tests and we create valid and invalid strings. No ground string is used, so the notion of killing mutants does not apply to mutating grammars. Four mutation operators for grammars are defined below.

1. ***Nonterminal Replacement***:

   | Every nonterminal symbol in a production is replaced by other nonterminal symbols. |
   |---|

This is a very broad mutation operator that could result in many strings that are not only invalid, they are so far away from valid strings that they are useless for testing. If the grammar provides specific rules or syntactic restrictions, some nonterminal replacements can be avoided. This is analogous to avoiding compiler errors in program-based mutation. For example, some strings represent type structures and only nonterminals of the same or compatible type should be replaced.

The production `dep ::= "deposit" account amount` can be mutated to create the following productions:

```
dep ::= "deposit" amount amount
dep ::= "deposit" account digit
```

Which can result in the following tests:

```
deposit $19.22 $12.35
deposit 739 1
```

2. ***Terminal Replacement***:

   | Every terminal symbol in a production is replaced by other terminal symbols. |
   |---|

Just as with nonterminal replacement, some terminal replacements may not be appropriate. Recognizing them depends on the particular grammar that is being mutated. For example, the production `amount ::= "$" digit+ "." digit2` can be mutated to create the following three productions:

```
amount ::= "." digit+ "." digit2
amount ::= "$" digit+ "$" digit2
amount ::= "$" digit+ "1" digit2
```

Which can result in the corresponding tests:

```
deposit 739 .12.35
deposit 739 $12$35
deposit 739 $12135
```

3. ***Terminal and Nonterminal Deletion***:
   Every terminal and nonterminal symbol in a production is deleted.

For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three productions:

```
dep ::= account amount
dep ::= "deposit" amount
dep ::= "deposit" account
```

Which can result in the corresponding tests:

```
739 $12.35
deposit $12.35
deposit 739
```

4. ***Terminal and Nonterminal Duplication***:
   Every terminal and nonterminal symbol in a production is duplicated.

This is sometimes called the "stutter" operator. For example, the production `dep ::= "deposit" account amount` can be mutated to create the following three mutated productions:

```
dep ::= "deposit" "deposit" account amount
dep ::= "deposit" account account amount
dep ::= "deposit" account amount amount
```

Which can result in the corresponding tests:

```
deposit deposit 739 $12.35
deposit 739 739 $12.35
deposit 739 $12.35 $12.35
```

We have significantly more experience with program-based mutation operators than grammar-based operators, so this list should be treated as being much less definitive.

These mutation operators can be applied in either of two ways. One is to mutate the grammar and then generate inputs. The other is to use the correct grammar, but one time during each derivation apply a mutation operator to the production being used. The operators are typically applied during production, because the resulting inputs are usually "closer" to valid inputs than if the entire grammar is corrupted. This approach is used in the previous examples.

Just as with program-based mutation, some inputs from a mutated grammar rule are still in the grammar. The example above of changing the rule

```
dep     ::= "deposit" account amount
```

to be

```
dep     ::= "debit" account amount
```

yields an "equivalent" mutant. The resulting input, `debit 739 $12.35`, is a valid input, although the effects are (sadly) quite different for the customer. If the idea is to generate invalid inputs exclusively, some way must be found to screen out mutant inputs that are valid. Although this sounds much like the equivalence problem for programs, the difference is small but significant. Here the problem is solvable and can be solved by creating a recognizer from the grammar, and checking each string as it is produced.

Many programs are supposed to accept some, but not all, inputs from a larger language. Consider the example of a web application that allows users to provide reviews. For security reasons the application should restrict its inputs to a subset of HTML; otherwise a malicious reviewer can enter a "review" that also uses HTML to implement an attack such as redirecting a user to a different website. From a testing perspective, we have two grammars: the full HTML grammar, and a grammar for the subset. Invalid tests that are in the first grammar, but not the susbet, are good tests because they can represent an attack.

### XML Example

Section 9.5.1 showed examples of generating tests in the form of XML messages from a schema grammar definition. It is also convenient to apply mutation to XML schemas to produce invalid messages. Some programs will use XML parsers that validate the messages against the grammar. If they do, it is likely that the software will usually behave correctly on invalid messages, but testers still need to verify this. If a validating parser is not used, this can be a rich source for programming mistakes. It is also fairly common for programs to use XML messages without having an explicit schema definition. In this case, it is very helpful for the test engineer to develop the schema as a first step to developing tests.

XML schemas have a rich collection of built-in datatypes, which come with a large number of *constraining facets*. In XML, *constraining facets* are used to restrict further the range of values. The example in Figure 9.9 uses several constraining facets, including *fractionDigits*, *minInclusive*, and *minOccurs*. This suggests further mutation operators for XML schemas that modify the **values** of facets. This can often result in a rich collection of tests for software that use inputs described with XML.

Given the following four lines in the books schema in Figure 9.9:

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="0"/>
<xs:totalDigits value="4"/>
<xs:pattern value="[0-9]{10}"/>
```

we might construct the mutants:

```
<xs:element name="ISBN" type="xs:isbnType" minOccurs="1"/>

<xs:element name="price" type="xs:decimal" fractionDigits="1" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="3" minInclusive="0"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" minInclusive="1"/>
<xs:element name="price" type="xs:decimal" fractionDigits="2" maxInclusive="0"/>

<xs:totalDigits value="5"/>
<xs:totalDigits value="0"/>

<xs:pattern value="[0-8]{10}"/>
<xs:pattern value="[1-9]{10}"/>
<xs:pattern value="[0-9]{9}"/>
```

---

## Exercises, Section 9.5.

1. Generate tests to satisfy TSC for the bank example grammar based on the BNF in Section 9.5.1. Try **not** to satisfy PDC.

2. Generate tests to satisfy PDC for the bank example grammar.

3. Consider the following BNF with start symbol A:

   ```
   A ::= B"@"C"."B
   B ::= BL | L
   C ::= B | B"."B
   L ::= "a" | "b" | "c" | ...  | "y" | "z"
   ```

   and the following six possible test cases:

   ```
   t1 = a@a.a
   t2 = aa.bb@cc.dd
   t3 = mm@pp
   t4 = aaa@bb.cc.dd
   t5 = bill
   t6 = @x.y
   ```

   For each of the six tests, (1) identify the test sequence as either "in" the BNF, and give a derivation, or (2) identify the test sequence as "out" of the BNF, and give a mutant derivation that results in that test. (Use only one mutation per test, and use it only one time per test.)

4. Provide a BNF description of the inputs to the *cal()* method in the homework set for Section 9.2.2. Succinctly describe any requirements or constraints on the inputs that are hard to model with the BNF.

5. Answer questions (a) through (c) for the following grammar.

```
val    ::= number | val pair
number ::= digit+
pair   ::= number op | number pair op
op     ::= "+" | "-" | "*" | "/"
digit  ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
```

Also consider the following mutated version, which adds an additional rule to the grammar:

```
pair  ::= number op | number pair op | op number
```

(a) Which of the following strings can be generated by the (unmutated) grammar?

```
42
4 2
4 + 2
4 2 +
4 2 7 - *
4 2 - 7 *
4 2 - 7 * +
```

(b) Find a string that is generated by the mutated grammar, but not by the original grammar.

(c) (*Challenging*) Find a string whose generation uses the new rule in the mutant grammar, but is also in the original grammar. Demonstrate your answer by giving the two relevant derivations.

6. Answer questions (a) and (b) for the following grammar.

```
phoneNumber   ::= exchangePart dash numberPart
exchangePart  ::= special zeroOrSpecial other
numberPart    ::= ordinary4
ordinary      ::= zero | special | other
zeroOrSpecial ::= zero | special
zero          ::= "0"
special       ::= "1" | "2"
other         ::= "3" | "4" | "5" | "6" | "7" | "8" | "9"
dash          ::= "-"
```

(a) Classify the following as either phoneNumbers (or not). For non-phone numbers, indicate the problem.

- 123-4567
- 012-3456
- 109-1212
- 246-9900
- 113-1111

(b) Consider the following mutation of the grammar:

```
exchangePart ::= special  ordinary  other
```

If possible, identify a string that appears in the mutated grammar but not in the original grammar, another string that is in the original but not the mutated, and a third string that is in both.

7. Use the web application program *calculate* to answer the following questions. *calculate* is on the second author's website (at *http://cs.gmu.edu:8080/offutt/servlet/calculate* as of this writing).

   (a) Analyze the inputs for *calculate* and determine and write the grammar for the inputs. You can express the grammar in BNF, an XML schema, or another form if you think it's appropriate. Submit your grammar.
   (b) Use the mutation ideas in this chapter to generate tests for *calculate*. Submit all tests; be sure to include expected outputs.
   (c) Automate your tests using a web testing framework such as HttpUnit or Selenium. Submit screen printouts of any anomalous behavior.

8. Java provides a package, *java.util.regex*, to manipulate regular expressions. Write a regular expression for URLs and then evaluate a set of URLs against your regular expression. This assignment involves programming, since input structure testing without automation is pointless.

   (a) Write (or find) a regular expression for a URL. Your regular expression does not need to be so general that it accounts for every possible URL, but give your best effort (for example `"*"` will not be considered a good effort). You are strongly encouraged to do some web surfing to find some candidate regular expressions. One suggestion is to visit the *Regular Expression Library*.
   (b) Collect a set of URLs from a small web site (such as a set of course web pages). Your set needs to contain at least 20 (different) URLs. Use the *java.util.regex* package to validate each URL against your regular expression.
   (c) Construct a valid URL that is not valid with respect to your regular expression (and show this with the appropriate *java.util.regex call*). If you have done an outstanding job in part 1, explain why your regular expression does not have any such URLs.

9. Why is the equivalent mutant problem solvable for BNF grammars but not for program-based mutation? (Hint: The answer to this question is based on some fairly subtle theory.)

---

## 9.6 Bibliographic Notes

We trace the use of grammars for testing compilers back to Hanford in 1972 [27], who motivated subsequent related work [6, 21, 33, 48, 49]. Maurer's Data Generation Language (DGL) tool [42] showed the applicability of grammar-based generation to many types of software, a theme echoed in detail by Beizer [7]. A recent paper was published by Guo and Qiu [24].

Legend has it that the first ideas of mutation analysis were postulated in 1971 in a class term paper by Richard Lipton. The first research papers were published by Budd and Sayward [11], Hamlet [25], and DeMillo, Lipton, and Sayward [19] in the late 1970s; DeMillo, Lipton, and Sayward's paper [19] is generally cited as the seminal reference. Mutation has primarily been applied to software by creating mutant versions of the source, but has also been applied to other languages, including formal software specifications.

The original analysis of the number of mutants was by Budd [12], who analyzed the number of mutants generated for a program and found it to be roughly proportional to the product of the number of variable references times the number of data objects ($O(Refs * Vars)$). A later analysis [2] claimed that the number of mutants is $O(Lines*Refs)$–assuming that the number of data objects in a program is proportional to the number of lines. This was reduced to $O(Lines * Lines)$ for most programs; this figure appears in most of the literature.

A statistical regression analysis of actual programs by Offutt et al. [44] showed that the number of lines did **not** contribute to the number of mutants, but that Budd's figure is accurate. The selective mutation approach mentioned below under "Designing Mutation Operators" eliminates the number of data objects so that the number of mutants is proportional to the number of variable references ($O(Refs)$).

Weak mutation has been widely discussed [23, 31, 53, 45], and experimentation has shown that the difference is very small [30, 41, 45]. Mutation operators have been designed for various programming languages, including Fortran IV [5, 15], COBOL [28], Fortran 77 [20, 37], C [17], C integration testing [16], Lisp [14], Ada [9, 47], Java [36], and Java class relationships [39, 40].

Research proof-of-concept tools have been built for Fortran IV and 77, COBOL, C, Java, and Java class relationships. One of the most widely used tools was Mothra [18, 20], a mutation system for Fortran 77 that was built in the mid-80s at Georgia Tech. Mothra was built under the leadership of Rich DeMillo, with most of the design done by DeMillo and Offutt, and most of the implementation by Offutt and King, with help from Krauser and Spafford. In its heyday in the early '90s, Mothra was installed at well over a hundred sites and the research that was done to build Mothra and that later used Mothra as a laboratory resulted in around half a dozen PhD dissertations and many dozens of papers. A more recent tool for Java is muJava [40, 46], which supports both statement level and object-oriented mutation operators, and accepts tests written in JUnit. muJava has been used to support hundreds of testing research projects. As far as we know, the only commercial tool that supports mutation is by the company Certess [26], in the chip design industry.

The coupling effect says that complex faults are coupled to simple faults in such a way that test data that detects all simple faults will detect most complex faults [19]. The coupling effect was supported empirically for programs in 1992 [43], and has shown to hold probabilistically for large classes of programs in 1995 [51]. Budd [13] discussed the concept of program neighborhoods. The neighborhood concept was used to present the competent programmer hypothesis [19]. The fundamental premise of mutation testing, as coined by Geist et al. [22], is: **In practice, if the software contains a fault, there will usually**

**be a set of mutants that can be killed only by a test case that also detects that fault**.

The operation of replacing each statement with a "bomb" was called Statement ANalysis (SAN) in Mothra [37]. Mothra's Relational Operator Replacement (ROR) operator replaces each occurrence of a relational operator ($<$, $>$, $\leq$, $\geq$, $=$, $\neq$) with each other operator and the expression with *true* and *false*. The subsumption proofs in Section 9.2.2 used only the latter operators. Mothra's Logical Connector Replacement (LCR) operator replaces each occurrence of one of the logical operators ($\wedge$, $\vee$, $\equiv$, $\neq$) with each other operator and the entire expression with $true$, $false$, $leftop$ and $rightop$. $leftop$ and $rightop$ are special mutation operators that return the left side and the right side, respectively, of a relational expression. The mutation operator that removes each statement in the program was called Statement DeLetion (SDL) in Mothra [37] and muJava.

Several authors [3, 4, 8, 50, 52] have used traces from model checkers to generate tests, including mutation based tests. The text from Huth and Ryan [32] provides an easily accessible introduction to model checking and discusses use of the SMV system.

Jia and Harman published a thorough review of the mutation testing literature in 2010 [35].

One of the key technologies being used to transmit data among heterogeneous software components on the Web is the eXtensible Markup Language (XML) [1, 10]. Data-based mutation defines **generic classes** of mutation operators. These mutation operator classes are intended to work with different grammars. The current literature [38] cites operator classes that modify the length of value strings and determine whether or not a value is in a pre-defined set of values.

# Bibliography


[1] W3C #28. Extensible markup language (XML) 1.0 (second edition)-W3C recommendation, October 2000. http://www.w3.org/XML/#9802xml10.

[2] Alan T. Acree, Tim A. Budd, Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[3] Paul Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. *International Journal of Quality, Reliability, and Safety Engineering*, 8(4):1–26, December 2000.

[4] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54, Brisbane, Australia, December 1998.

[5] D. M. St. Andre. Pilot mutation system (PIMS) user's manual. Technical report GIT-ICS-79/04, Georgia Institute of Technology, April 1979.

[6] J. A. Bauer and A. B. Finger. Test plan generation using formal grammars. In *Fourth International Conference on Software Engineering*, pages 425–432, Munich, September 1979.

[7] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, Inc, New York NY, 2nd edition, 1990. ISBN 0-442-20672-0.

[8] Paul Black, Vladim Okun, and Y. Yesha. Mutation operators for specifications. In *Fifteenth IEEE International Conference on Automated Software Engineering*, pages 81–88, September 2000.

[9] John H. Bowser. Reference manual for Ada mutant operators. Technical report GIT-SERC-88/02, Georgia Institute of Technology, February 1988.

[10] T. Bray, J. Paoli, and C. M. Sperberg-McQueen. Extensible markup language (XML) 1.0. W3C recommendation, February 1998. http://www.w3.org/TR/REC-xml/.

[11] Tim Budd and Fred Sayward. Users guide to the Pilot mutation system. Technical report 114, Department of Computer Science, Yale University, 1977.

[12] Tim A. Budd. *Mutation Analysis of Program Test Data*. PhD thesis, Yale University, New Haven CT, 1980.

[13] Tim A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, November 1982.

[14] Tim A. Budd and Richard J. Lipton. Proving lisp programs using test data. In *Digest for the Workshop on Software Testing and Test Documentation*, pages 374–403, Ft. Lauderdale FL, December 1978. IEEE Computer Society Press.

[15] Tim A. Budd, Richard J. Lipton, Richard A. DeMillo, and Fred G. Sayward. Mutation analysis. Technical report GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, April 1979.

[16] Márcio Delamaro, José C. Maldonado, and Aditya P. Mathur. Interface mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, 27(3):228–247, March 2001.

[17] Márcio E. Delamaro and José C. Maldonado. Proteum-A tool for the assessment of test adequacy for C programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, New Brunswick, NJ, July 1996.

[18] Richard A. DeMillo, Dany S. Guindi, Kim N. King, W. Michael McCracken, and Jeff Offutt. An extended overview of the Mothra software testing environment. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, Banff, Alberta, July 1988. IEEE Computer Society Press.

[19] Richard A. DeMillo, Richard J. Lipton, and Fred G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[20] Richard A. DeMillo and Jeff Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[21] A. G. Duncan and J. S. Hutchison. Using attributed grammars to test designs and implementations. In *Proceedings of the 5th International Conference on Software Engineering (ICSE 5)*, pages 170–177, San Diego, CA, March 1981. IEEE Computer Society Press.

[22] Robert Geist, Jeff Offutt, and Fred Harris. Estimation and enhancement of real-time software reliability through mutation analysis. *IEEE Transactions on Computers*, 41(5):550–558, May 1992. Special Issue on Fault-Tolerant Computing.

[23] M. R. Girgis and M. R. Woodward. An integrated system for program testing using weak mutation and data flow analysis. In *Proceedings of the Eighth International Conference on Software Engineering*, pages 313–319, London UK, August 1985. IEEE Computer Society Press.

[24] Hai-Feng Guo and Zongyan Qiu. Automatic grammar-based test generation. In *Testing Software and Systems*, volume LNCS 8254, pages 17–32. Springer-Verlag, 2013.

[25] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering*, 3(4):279–290, July 1977.

[26] Mark Hampton and Stephane Petithomme. Leveraging a commercial mutation analysis tool for research. In *Third IEEE Workshop on Mutation Analysis (Mutation 2007)*, pages 203–209, Windsor, UK, September 2007.

[27] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 4:242–257, 1970.

[28] J. M. Hanks. Testing COBOL programs by mutation: Volume I-introduction to the CMS.1 system, volume II - CMS.1 system documentation. Technical report GIT-ICS-80/04, Georgia Institute of Technology, February 1980.

[29] Mark Harman, Yue Jia, and William B. Langdon. How higher order mutation helps mutation testing (keynote). In *5th International Workshop on Mutation Analysis (Mutation 2010)*, Paris, France, 2010.

[30] J. R. Horgan and Aditya P. Mathur. Weak mutation is probably strong mutation. Technical report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette IN, December 1990.

[31] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[32] Michael Huth and Mark D. Ryan. *Logic in Computer Science: Modelling and Reasoning About Systems*. Cambridge University Press, Cambridge, UK, 2000.

[33] D. C. Ince. The automatic generation of test data. *The Computer Journal*, 30(1):63–69, February 1987.

[34] Yue Jia and Mark Harman. Constructing subtle faults using higher order mutation testing. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2008)*, pages 249–258, Beijing, September 2008.

[35] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions of Software Engineering*, 37(5):649–678, September 2011.

[36] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the effectiveness of object-oriented strategies with the mutation method. In *Proceedings of Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 4–100, San Jose, CA, October 2000. Wiley's Software Testing, Verification, and Reliability, December 2001.

[37] Kim N. King and Jeff Offutt. A Fortran language system for mutation-based software testing. *Software-Practice and Experience*, 21(7):685–718, July 1991.

[38] Suet Chun Lee and Jeff Offutt. Generating test cases for XML-based Web component interactions using mutation analysis. In *Proceedings of the 12th International Symposium on Software Reliability Engineering*, pages 200–209, Hong Kong China, November 2001. IEEE Computer Society Press.

[39] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*, pages 352–363, Annapolis MD, November 2002. IEEE Computer Society Press.

[40] Yu-Seung Ma, Jeff Offutt, and Yong-Rae Kwon. MuJava : An automated class mutation system. *Software Testing, Verification, and Reliability, Wiley*, 15(2):97–133, June 2005.

[41] B. Marick. The weak mutation hypothesis. In *Proceedings of the Fourth Symposium on Software Testing, Analysis, and Verification*, pages 190–199, Victoria, British Columbia, Canada, October 1991. IEEE Computer Society Press.

[42] Peter M. Maurer. Generating testing data with enhanced context-free grammars. *IEEE Software*, 7(4):50–55, July 1990.

[43] Jeff Offutt. Investigations of the software testing coupling effect. *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992.

[44] Jeff Offutt, Ammei Lee, Gregg Rothermel, Roland Untch, and Christian Zapf. An experimental determination of sufficient mutation operators. *ACM Transactions on Software Engineering Methodology*, 5(2):99–118, April 1996.

[45] Jeff Offutt and Stephen D. Lee. An empirical evaluation of weak mutation. *IEEE Transactions on Software Engineering*, 20(5):337–344, May 1994.

[46] Jeff Offutt, Yu-Seung Ma, and Yong-Rae Kwon. muJava home page. Online, 2005. http://cs.gmu.edu/∼offutt/mujava/, last access July 2014.

[47] Jeff Offutt, Jeffrey Payne, and Jeffrey M. Voas. Mutation operators for Ada. Technical report ISSE-TR-96-09, Department of Information and Software Engineering, George Mason University, Fairfax VA, March 1996. http://www.cs.gmu.edu/∼tr_admin/.

[48] A. J. Payne. A formalised technique for expressing compiler exercisers. *Sigplan Notices*, 13(1):59–69, January 1978.

[49] P. Purdom. A sentence generator for testing parsers. *BIT*, 12:366–375, July 1972.

[50] S. Rayadurgam and M. P. E. Heimdahl. Coverage based test-case generation using model checkers. In *8th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, pages 83–91, April 2001.

[51] K. S. How Tai Wah. Fault coupling in finite bijective functions. *Software Testing, Verification, and Reliability, Wiley*, 5(1):3–47, March 1995.

[52] Duminda Wijesekera, Lingya Sun, Paul Ammann, and Gordon Fraser. Relating counterexamples to test cases in CTL model checking specifications. In *A-MOST '07: Third ACM Workshop on the Advances in Model-Based Testing, co-located with ISSTA 2007*, London, UK, July 2007.

[53] M. R. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, Banff, Alberta, July 1988. IEEE Computer Society Press.

# Index