A Mutation Testing Tool for Java Programs
Ett verktyg för mutationstestning av Javaprogram
Examensarbete i Datalogi, 20p

Mattias Bybro
Handledare: Prof. Karl Meinke
Examinator: Prof. Stefan Arnborg

August 15, 2003

Abstract

High quality software can not be done without high quality testing. Mutation testing measures how “good” our tests are by inserting faults into the program under test. Each fault generates a new program, a mutant, that is slightly different from the original. The idea is that the tests are adequate if they detect all mutants. This thesis relates previous work on the theoretical foundation of mutation testing and emphasizes the role of efficient mutation operators. Mutation operators determine what types of faults to use. We have investigated the properties of an adequate and efficient set of mutation operators. Also, a prototype of a mutation testing tool for Java aimed at real software projects was implemented. This tool was used to demonstrate how the properties of efficient mutation operators allow us to find redundant mutants. Not only do these results show how to reduce the cost of mutation testing but they also hint at a more theoretical approach to design sets of mutation operators. Further steps towards a complete mutation testing system are suggested and we conclude that a usable tool for the software industry is within reach.

Sammanfattning
Ett Verktyg för Mutationstestning av Javaprogram

Mjukvara av hög kvalitet kan inte konstrueras utan tester av hög kvalitet. Mutationstesting ger ett mått på hur ”bra” våra tester är genom att införa fel i programmet som testas. Varje fel ger upphov till ett nytt program, en mutant, vilket skiljer sig något från det ursprungliga programmet. Tanken är att testerna är adekvata om de kan hitta alla mutanter. Detta examensarbete återger den teoretiska grunden för mutationstestning och betonar vikten av effektiva mutationsoperatorer. Mutationsoperatorer bestämmer vilka slags fel som förs in. Vi har undersökt egenskaperna hos en adekvat och effektiv mängd av mutationsoperatorer. Dessutom har vi implementerat en prototyp av ett mutationstestningsverktyg för Java som är riktat mot verkliga mjukvaruprojekt. Detta verktyg användes för att visa hur egenskaperna hos effektiva mutationsoperatorer tillåter oss att hitta mutanter som inte leder till högre testkvalitet. Förutom att minska kostnaderna för mutationstestning ger detta experiment en ledtråd till ett mer teoretiskt tillvägagångssätt för att designa mutationsoperatorer. I arbetet lägger vi fram förslag på nästa steg för att skapa ett komplett system för mutationstestning. Slutsatsen är att ett användbart verktyg för mjukvaruindustrin är inom räckhåll.

Preface

This Master's Thesis actually started a couple of years ago. I had written a module of a larger system and it was critical that it would not fail. To prove that it would not is indeed a very difficult task. Hence I started the search for tools and methods to test and improve the reliability of the program. Soon, I was reading a web page describing this very promising "mutation testing" method. It was a most natural choice to write my own thesis on this topic.

To read the report, some familiarity with mathematical concepts such as set theory and logic is recommended. Chapter 2 gives an introduction to the matematical and intuitive foundations of the theory, chapter 3 shows several (previous) ideas of how to describe the "coupling effect" and an attempt to summarize them matematically. These results are used to deduce properties of a set of efficient mutation operators. The brevity of chapter 4 does not quite reveal the amount of work that was invested in creating a mutation testing tool for Java, but some basic techniques to make such a tool are presented. The future of mutation testing is discussed in chapter 5.

This Computer Science thesis was supervised by Prof. Karl Meinke at the department of Numerical Analysis and Computer Science, Nada, at the Royal Institute of Technology, KTH. I would like to use a couple of lines to thank Karl for reviewing and finding all those missing definitions and thoughts, for showing me how to fix it and for his keen interest in solving the unsolvable (software testing) and new/old ideas how to do it (like mutation testing).

Mattias Bybro
Stockholm, June 2003

Contents

1 Introduction **1**

2 Mutation Testing **3**
- 2.1 Mutation Operators 6
- 2.2 Equivalent Mutants 7
  - 2.2.1 Automatically Detecting Equivalent Mutants 7

3 The Coupling Effect **9**
- 3.1 Empirical Studies of Higher-order Mutants 9
- 3.2 Theoretical Study of the Coupling Effect 10
- 3.3 Semantic Size 10
- 3.4 Failure Regions and Mutation Operators 11
  - 3.4.1 An Efficient Set of Mutation Operators 14
  - 3.4.2 What does Mutation Adequacy Measure? 15

4 Implementing a Mutation Testing System for Java and JUnit **17**
- 4.1 Metamutant Generator 19
  - 4.1.1 Java Mutation Operators 20
- 4.2 Equivalence Detector 20
- 4.3 Test Runner 22
- 4.4 Testing the Triangle Program 22
  - 4.4.1 Equivalent Mutants 22
  - 4.4.2 Experiment 1: Mutation Adequacy 23
  - 4.4.3 Experiment 2: Coupling Effect Ratio 24

5 Conclusions **26**
- 5.1 Mutation Operators 26
  - 5.1.1 The Mutation Testing Tool 27
- 5.2 Future Work 27

Bibliography **28**

Appendix: The TRIANGLE program **31**

Chapter 1

Introduction

$59,500,000,000! That is how much software bugs cost in the United States in the year 2002 alone, according to a recent NIST report [NI02]. The report estimates that $22,200,000,000 could be attributed to insufficient testing.

[SQAT] lists some recent major computer system failures caused by software bugs. Here are a few of them:

- 1996: The ESA Ariane 5 rocket fails due to a simple unchecked type error.
- 1999: The $125 million NASA Mars Climate Orbiter was believed to be lost in space. Why? Some data in the spacecraft software was used in English units. It should have been in metric units.
- 2002: Software bugs in Britain's national tax system resulted in more than 100,000 erroneous tax overcharges. Difficulties with integration testing was a major culprit.

Does the future look brighter? Methodologies like extreme programming have emphasised software quality and as the complexity of many software projects grows, software development processes are forced into more testing and quality assurance. Yet, testing is considered boring by many software developers and there is still resistance to accept important software engineering concepts and methods such as unit-testing, design-by-contract and daily or continuous builds.

Obviously, there is a great need for improved software quality and testing processes. How to get there is easier said than done, and this thesis do not aim to find the ultimate pesticide to eliminate all those bugs hidden in all those (spaghetti?) lines of code.

One important issue is the *coverage problem*, that is determining how well tested a piece of software is and when to stop. It is not unusual to stop testing when the project deadline is reached. Another common approach is to use some metric, usually *statement coverage* and set a threshold like “at least 80 % statement coverage”.

Then there is *mutation testing*, which is a coverage criterion that has its roots in the very definition of reliable test sets. This is what makes it fundamentally different from most other criteria.

Chapter 2

Mutation Testing

To estimate the number of fish of a certain species in a lake, one way to do it is letting some marked fish out in the lake (say, 20) and then catch some fish and count the marked ones. If we catch 40 fish and 4 of them are marked, then 1 out of 10 is marked and the population in the entire lake could be estimated to about 200. If we catch all marked fish, we would as a side-effect end up with almost the entire population in our nets.

Fault-based testing does something similar. We let some "marked" bugs loose in the code and try to catch them. If we catch them all, our "net" probably caught many of the other, fishier, fish. The unknown bugs, that is.

One of the fault-based testing strategies is *mutation testing.* There are many variations of mutation testing such as weak mutation [HO82], interface mutation [DM96] and specification-based mutation testing [MR01]. The method described in this thesis is *strong mutation* testing, but the idea is the same for all of them, namely to "mutate" the original program under test.

To mutate a program, an error is put somewhere in the code. And just like the fish in the lake, we will try to catch it. A typical mutation would be to replace $<$ with $>$ in one and only one expression. Example: the program $P =$

```
1. if (x > 0)
2.     doThis();
3. if (x > 10)
4.     doThat();
```

A mutation of P would be (line 1)

```
1. if (x < 0)
2.     doThis();
3. if (x > 10)
4.     doThat();
```

Another mutation (line 3):

```
1. if (x > 0)
2.     doThis();
3. if (x < 10)
4.    doThat();
```

Now we have made several copies of P and introduced a single mutation into each copy. These copies are called *mutants*. Let D denote the input domain. Assume we have a passing test set, $T \subset D$, that is P satisfies or passes every test in T. To get a measure of its *mutation adequacy*, we run the test set against each mutation and count the number of mutants for which T fails. If T fails for a certain mutant, we call that mutant *killed*. The idea is that if T detects this fault (kills the mutant), it will detect real, unknown faults as well. If T kills all mutants, it potentially detects many unknown faults. Mutants that are not killed are called *alive* and mutants (denoted *mu*) such that $\forall x \in D, P.x = \mu.x$ are called *equivalent*. We will write $\mu \equiv P$ if the mutant μ is equivalent to P. $P.x$ represents the evaluation of the program P on the input x. *Mutation adequacy* or *mutation score* is defined as (number of killed mutations)/(total number of non-equivalent mutations) * 100 %.

Why would this method work? [BD80] makes two fundamental assumptions; (a) the *competent programmer hypothesis* and (b) the *coupling effect*.

The traditional approach to software testing is to find some subset T (called the test set) of the input domain D, such that

$$\forall x \in T, P.x = f(x) \rightarrow \forall x \in D, P.x = f(x), \tag{2.1}$$

where f is a functional specification of the program P. (This is called a *reliable* test set.) To be able to reach this conclusion, some exhaustive testing strategy would be necessary. This is too strong a conclusion and is proven to be an undecidable problem. That is why mutation testing weakens the above:

$$\begin{gathered}\text{either } P \text{ is “pathological” or}\\ \forall x \in T, P.x = f(x) \rightarrow \forall x \in D, P.x = f(x)\end{gathered} \tag{2.2}$$

What is a “pathological” program?

$$P \text{ is “pathological”} \leftrightarrow P \notin \Phi, \tag{2.3}$$

where Φ is the set of programs in a “neighbourhood” of a correct program $P^\star$ satisfying f. Intuitively P is “pathological” if P is “far” from $P^\star$. We expect programmers to be competent enough to produce programs in this neighbourhood. (See figure 2.1.)

We can now reformulate 2.1 to

$$\begin{gathered}\forall x \in T, P.x = f(x) \wedge\\ \forall Q \in \Phi\ (Q \equiv P \vee \exists x \in T, Q.x \neq P.x)\\ \rightarrow \forall x \in D, P.x = f(x),\end{gathered} \tag{2.4}$$

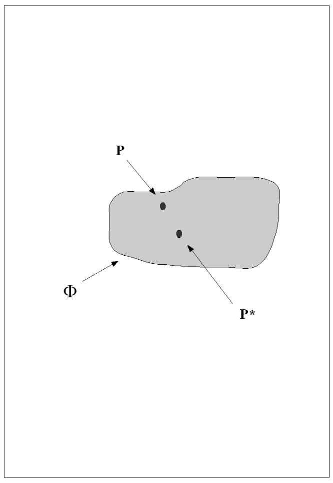

Figure 2.1: The competent programmer hypothesis assumes that programmers write programs in the neighbourhood Φ of the correct program $P^{\star}$. P is such a program. The area within the frame represents the set of all programs.

i. e. if P passes all tests in T and all non-equivalent mutants are killed by some test in T, then P satisfies T. If we find T that satisfies the above criteria, then we say that "P passes the Φ mutant test". A test point $x \in T$ such that $P.x \neq Q.x$ *differentiates* P from Q.

The competent programmer hypothesis assumes that the program under test is not "pathological" and allows us to limit the problem. Instead of exhaustively testing P with an enormous (almost infinite) test set, we could find all programs in the neighbourhood of P and feed each of them with some test values to differentiate them from P. Although this limits the problem, it is also a huge and indeed impossible task.

However, the coupling effect says there is a small subset μ of Φ, such that if P passes the μ mutant test $\rightarrow$ P passes the Φ mutant test. All we have to do is find μ. In the next section, we will look at an instance of μ which is used to mutate FORTRAN programs. (You have already seen in the introductory example how a program in this μ might be like.)

First we will compile a list of problems with mutation testing:

- **The coupling effect.** How can we be sure that it holds? Does simple syntactic changes like the ones on page 3 really define an appropriate μ set?

- **Equivalent mutants.** Can we tell when $Q \equiv P$? If we do not find any test point that can differentiate the two, is this because they are equivalent or is it because it is hard to kill the mutant?

Table 2.1: FORTRAN mutation operators. (*Source: [OL93].*)

Operator	Description
AAR	*A*rray reference for *A*rray reference *R*eplacement
ABS	*ABS*solute value insertion
ACR	*A*rray reference for *C*onstant *R*eplacement
AOR	*A*rithmetic *O*perator *R*eplacement
ASR	*A*rray reference for *S*calar variable *R*eplacement
CAR	*C*onstant for *A*rray reference *R*eplacement
CNR	*C*omparable array *N*ame *R*eplacement
CRP	*C*onstants *ReP*lacement
CSR	*C*onstant for *S*calar variable *R*eplacement
DER	*D*o statement *E*nd *R*eplacement
DSA	*D*ata *S*tatement *A*lterations
GLR	*G*oto *L*abel *R*eplacement
LCR	*L*ogical *C*onnector *R*eplacement
ROR	*R*elational *O*perator *R*eplacement
RSR	*R*eturn *S*tatement *R*eplacement
SAN	*S*tatement *AN*alysis
SAR	*S*calar for *A*rray reference *R*eplacement
SCR	*S*calar for *C*onstant *R*eplacement
SDL	*S*tatement *DeL*etion
SRC	*S*ou*R*ce *C*onstant replacement
SVR	*S*calar *V*ariable *R*eplacement
UOI	*U*nary *O*perator *I*nsertion

- **Expensive testing.** One might suspect that μ is quite a large set and thus executing and creating tests will be costly.

2.1 Mutation Operators

For real-world programs, μ is defined as the set of programs generated by applying *mutation operators* to the program under test, P. What is a mutation operator? A mutation operator is a simple syntactic or semantic transformation rule, like replacing $<$ with $>$. (Table 2.1 defines some FORTRAN mutation operators.)

What would constitute a good set of mutation operators? Before answering that, we will need to look at some previous results on this (see chapter 3). For now, we just conclude that the validity of the coupling effect is highly dependent upon the set of mutation operators.

For instance, if all mutations always cause the test to fail by throwing an exception, then the mutation adequacy score will be high, but the fault detection abilities low.

2.2 Equivalent Mutants

There is also the problem with equivalent mutants. If $\forall x \in D, Q.x = P.x$, then the mutant Q is equivalent to the program P. An example of an equivalent mutant: $P =$

```
1. if (x > 0)
2.    if (y > x)
3.       doThis();
4.    fi
5. fi
```

$Q =$ (applied ABS mutation operator to line 2)

```
1. if (x > 0)
2.    if (y > abs(x))
3.       doThis();
4.    fi
5. fi
```

Since $x = |x|$ for $x > 0$, then Q is an equivalent mutant of P. To manually detect and remove an equivalent mutant is very time consuming and prone to error. Therefore automatic detection algorithms have been developed [OC94, OP97], estimated to remove approximately 50 % of the equivalent mutants. How common are equivalent mutants? Of course, there is no universal rule to this, but in the sample programs used by Offutt and Pan in [OP97], 9 % of all mutants were equivalent. The results presented in this thesis (see section 4.4.2) are similar; 8 % of the mutants were equivalent. Note that some mutation operators generate more equivalent mutants; the ABS operator accounted for half the equivalent mutants.

2.2.1 Automatically Detecting Equivalent Mutants

It is an undecidable problem to determine whether two programs are equal. For simple mutations of programs, the situation is slightly different. Consider the example in the previous section. Could we not create some smart software that knew $abs(x) = x$ if $x \geq 0$?

In some cases, we can. [OP97] uses constraint resolving to identify equivalent mutants. This is implemented as a specialized theorem prover based on simple logical relations like

$$x > 0 \Rightarrow x = |x| \tag{2.5}$$

The goal is to find a contradiction in the set of constraints for a statement. Since the constraints are independent of test values, we know that the mutant is equivalent if we find a contradiction.

What does a set of constraints look like? In the above unmutated code snippet, the set of constraints in line 3, C_p, would be $(y > x) \wedge (x > 0)$ and for the mutated code, C_μ, $(y > |x|) \wedge (x > 0)$. From these, we can build a *necessity constraint system*, C_n. This is based on the *necessity condition* of a test set. If this condition is to be met, the state of the mutated program immediately after the execution of the mutated statement must be different from the state of the unmutated program at the same point. If the condition cannot be met, the mutant is equivalent.

We can derive C_n:

$$(|x| \neq x) \wedge (x > 0), \tag{2.6}$$

The system knows that $x > 0$ implies $x = |x|$ and that $(|x| \neq x) \leftrightarrow \neg(x = |x|)$. Contradiction, regardless of choice of x. Hence, the mutant is equivalent.

Having applied automatic equivalent detection and run the test set, we have killed most mutants and found some to be equivalent. The remaining mutants are called *stubborn*. They might be equivalent or they might not be. To assist the human tester in analyzing these mutants, [HH99] suggests program slicing to reduce a complex set of statements to a simpler one.

Chapter 3

The Coupling Effect

Of course, it is not possible to create the entire set of “non-pathological” programs, Φ. Could we find a small subset μ of Φ such that if P passes the μ mutant test, then will it also pass the Φ mutant test? Empirical [OF92] results and for some cases theoretical [WA00] studies have shown that there are indeed cases where this is true, but only for a strict definition of Φ. The following sections relate these results, give an explanation of what mutation testing really is testing and list the desired properties of an efficient set of mutation operators.

3.1 Empirical Studies of Higher-order Mutants

Offutt [OF92] investigated the *mutation coupling effect*, which is the coupling between first-order and higher-order mutants. First-order mutants are mutants created by inserting one single fault into the program. Higher-order mutants are created by inserting more than one fault into the program. Offutt then showed that a test set that kills all first-order mutants would kill almost all second-order mutants too. For a program similar to the one in the appendix, there were 951 first-order mutants and $350,982$ second-order mutants. An adequate test set killing all first-order mutants was constructed. This test set killed all except ten of the second-order mutants.

What does this result prove? Well, we cannot say that the coupling effect holds for the general case since we do not know whether higher-order mutants accurately model complex faults[1].

It does seem, however, that we could say something about the compound effect of several bugs. First assume that it is possible to isolate two faults. If we could detect each of these isolated faults with some strategy, the above result indicates that they will not cancel each other out and that we with great probability will detect them even if not isolated.

[1]At this point we would need to define “complex fault” too, but for now it will do to just think of it as “real-life bugs”.

This informal result will be formalised in the next section.

3.2 Theoretical Study of the Coupling Effect

Wah [WA00] models programs as finite functions. He defines the program as a function $f = g \circ h$, and two faulty versions thereof; $f' = g' \circ h$ and $f'' = g \circ h'$, where g' and h' denote faulty versions of g and h respectively. Define $f''' = g' \circ h'$. The question is: if we detect f' and f'', will we also detect f'''?

From a software specification, we know the expected behaviour of the correct program f. For a set of test points T, can we decide whether the program under test is correct or incorrect? Under what circumstances will we believe the program to be f when it actually is f'''? Only if for each $t \in T$, $f(t) = f'''(t)$.

If we have a single-point test set $T = \{t_0\}$ that detects f' and f'', then we will believe $f'''(t)$ to be f if this constraint holds:

$$f(t_0) = f'''(t_0) = (g' \circ h')(t_0) = g'(h'(t_0)) \tag{3.1}$$

To get a measure of the coupling effect, ignoring the fact that some functions (programs) are more likely to exist than others, we count the number of functions where the above constraint holds and compare it to the total number of possible functions. Note that the latter is a huge number; the class of programs with input and output domain of cardinality n will have n^n possible functions (programs), counting all equivalent programs as one.

Applying some combinatorial magic, the coupling effect ratio[2] for large n and a test set of order 1 will be $\approx 1 - \frac{1}{n}$. That is, if we have T detecting f' and f'', we will not detect f''' only in relatively few test cases.

We might think of f' and f'' as mutants. Their relation to mutation testing will be investigated in section 3.4.

3.3 Semantic Size

In [OH96], the *semantic size* of a fault is defined as "the relative size of the [input domain] D for which the output mapping is incorrect ... the semantic size would ideally be based on a usage distribution". We could rephrase this in mathematical terms:

$$Pr\{P.x \neq P_1.x\}, \qquad x \sim O(x) \tag{3.2}$$

That is, the probability that given an input value x from some usage distribution $O(x)$, the program P and the faulty program or mutation P_1 will produce different results.

[2]The ratio of test points detecting f' and f'' that also detects f'''. See definition in 3.4.

Offutt and Hayes tries to explain the coupling effect: The *failure region* of a fault is the portion of the input space that causes the fault to result in a failure[3]. Now, if for the failure region of every semantically large fault, there is a large overlap in the failure region for at least one small syntactic fault[4], the coupling effect holds.

Moreover, they hint at a possible intuitive explanation of why *selective mutation* would work. Selective mutation is when only some mutation operators or some mutants test the program as thoroughly as all of them. In this case, "all of them" refers to the 22 "standard" mutation operators often used in scientific papers. (They are listed in section 2.1.) There has been much work done finding a minimal set of mutation operators [OL93] and other efforts have been made to reduce the cost of mutation testing [MW93, HS90, WM97].

The idea is that selective mutation uses mutants having small semantic size. Although experiments showed that the semantic size was indeed smaller for the selected mutants than for the remaining ones – 31.60 % vs. 39.88 % – the evidence is inconclusive.

Finally, they state that semantically small mutants have the potential to lead to higher quality tests.

3.4 Failure Regions and Mutation Operators

To visualize failure regions, we define $D = (x, y, z)$, where x and y are integers in the interval $[1, 10]$ and $z = 5$, execute two mutants of the TRIANGLE program (see the appendix) and compare the output with the original, unmutated program P.

In figure 3.1 you see surface plots of the *diff function*

$$\Delta_{P,P_1}(x) = \begin{cases} 1, & if\ P.x \neq P_1.x \\ 0, & otherwise \end{cases}, \tag{3.3}$$

where P is the program under test and $P_1 \in \Phi$. The semantic size, s, (assuming uniform distribution of input values) would then be

$$s(P, P_1) = \frac{\sum\limits_{x \in D} \Delta_{P,P_1}(x)}{c(D)}, \tag{3.4}$$

where $c(D)$ is the cardinality of D. The failure region F is the set

$$F_{P,P_1} = \{x \mid \Delta_{P,P_1}(x) = 1\} \tag{3.5}$$

[3]IEEE definitions of *fault* and *failure*. Fault: An incorrect step, process, or data definition in a computer program. Failure: The inability of a system or component to perform its required functions within specified performance requirements.

[4]Note the difference between semantic and syntactic faults.

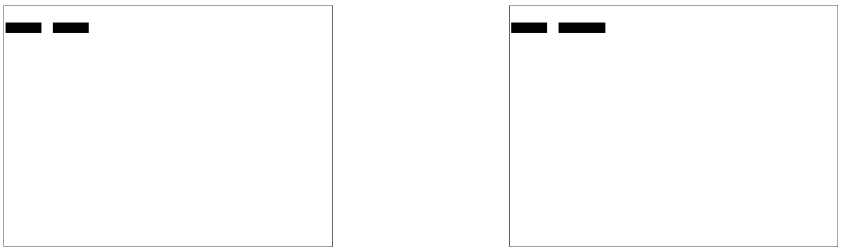

Figure 3.1: *Left:* failure region of Mutation 26, *right:* failure region of Mutation 27. The framed square is the input domain, the black areas indicate the failure region. *Failure region of Mutation 27:* Compare with mutation 26; the rightmost black area is slightly larger.

The coupling function δ for the two faulty versions P_1 and P_2 of P:

$$\delta_P^{P_1,P_2} = \begin{cases} 1, & if\ x \in F_{P,P_1}\ and\ x \in F_{P,P_2} \\ 0, & otherwise \end{cases} \tag{3.6}$$

Finally, we will define the coupling effect ratio of P_1 with respect to P_2:

$$CeR_P(P_1,P_2) = \frac{s(\delta_P^{P_1,P_2})}{s(\Delta_{P,P_1})} \tag{3.7}$$

In words, we could express this as an estimate of the probability of "a test point detecting P_1 also detecting P_2". Normally, we will not have the luxury to see the entire input domain at once. It will almost never be two-dimensional. Now we do, and it might serve as a conceptual model of how things work in higher dimensions. (See figures 3.1 and 3.2.)

Coupling of Mutation Operators: Consider $\delta_P^{\mu_i,\mu_j}$, where μ_i is the ith mutant. In the case shown in figure 3.1, where $i = 26$ and $j = 27$, $\delta_P^{\mu_i,\mu_j} = 1$ in just one point of the input domain, $(5,10)$. The coupling effect ratio $CeR_P^{\mu_{26},\mu_{27}} = 0.90$, implying that most faults detected by μ_{27} will also be detected by μ_{26}.

Detecting Bugs: Take a look at figure 3.2. Why is the buggy program P not detected (buggy with respect to a "perfect" program $P^\star$)? Because no test point is in the failure region of the bug. Consequently, the fact that $F_{P^\star,P} = F_{P,P^\star} \cap F_{P,\mu_x} \neq \oslash$ does not imply that we will detect the bug, given we have killed the mutant μ_x.

At first sight, this sounds discouraging. Why do we make all these efforts killing mutants if they do not detect real bugs? Because they often do. Again, the coupling effect comes to the rescue.

A program, P, and the "perfect" program $P^\star$ might be written as a sequence of functions:

$$\begin{aligned} P &= s_0 \circ \ldots \circ s_n, \\ P^\star &= s_0^\star \circ \ldots \circ s_n^\star \end{aligned} \tag{3.8}$$

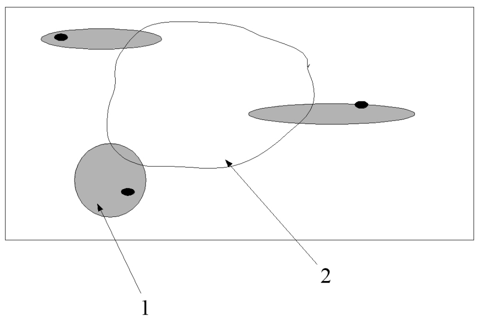

Figure 3.2: *An undetected bug.* Explanation to the figure: The area within the frame represents the input domain of the program under test. 1) Failure region for a mutant. 2) Failure region for an undetected bug. Black dots illustrate test points killing mutants.

The perfect program does not exist, but we assume we have a perfect (human?) test oracle that knows the value $P^\star(t)$ for any given t. The sequence $s_0^\star, \ldots, s_n^\star$ is just a representation of this knowledge. Define

$$\begin{aligned} P_i &= s_0 \circ \ldots \circ\ s_i^\star \circ \ldots\ \circ s_n, \\ \mu_i &= s_0 \circ \ldots \circ\ \mu_i(s_i) \circ \ldots\ \circ s_n, \end{aligned} \tag{3.9}$$

where $\mu_i(s_i)$ is some mutation of s_i. The coupling effect as explained by Wah in [WA00][5] shows that if we detect the first-order mutants of P, P_i, for $i = 1, \ldots, n$, then we will detect the n-order mutant $P^\star$ of P. That is, we can differ between the correct program $P^\star$ and the program under test P.

How do we detect P_i? Assume we detect μ_i. If we analyze the coupling effect ratio of μ_i with respect to P_i, there are three cases to consider:

- The coupling effect ratio $CeR_P^{\mu_i,P_i}$ is high. Detecting μ_i will probably detect B_i.
- The coupling effect ratio $CeR_P^{\mu_i,P_i}$ is low. Detecting μ_i will probably not detect B_i.
- $s_i = s_i^\star$. There is no failure region and hence no error in B at s_i. Had there been a bug highly coupled to a mutant in P, T would have detected it. Since we have a passing test set, no such bug was detected and consequently no such bug exists. That leaves us either with this case being true or with the bugs that are less coupled to mutants, the previous case.

[5]There are still holes in this theory; for example, the theory requires equal input domain size of each s_i. See [WA00] for a thorough discussion.

Conclusion: If the coupling effect ratio is high for each pair μ_i, P_i and we have a passing test set then $s_i = s_i^\star$ for each $i = 1, \ldots, n$. Our program under test, P, is correct[6].

The "only" thing we need to do is defining a set of mutation operators such that $CeR_P^{\mu_i,P_i}$ is high.

3.4.1 An Efficient Set of Mutation Operators

Let us revisit the earlier "$\delta_P^{\mu_i,\mu_j} = 1$ in only one point" issue. This indicates that the only bugs that μ_{27} will detect that μ_{26} with certainty will not, must have a failure region covering $(5, 10)$ and $F_{P,\mu_{26}} \cap F_{P,B} = \emptyset$. ($B$ is the potentially faulty program, P is a complete program specification which exists in this specific case, but generally does not.) Since this mutant probably will be killed by the same test case that kills μ_{26}, it is clear that it does not contribute much. This example shows there exists a coupling effect between mutants and explains the success of selective mutation.

Another example; a function computing $\sum_{i=1}^{10} i^2 \cdot f(i)$, applied mutation operators AAR and AOR, the former replacing each array reference with another and the latter replacing each arithmetic operator with another.

```
1. array[1..10] of integer squares =
      ( 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 );
2. array[1..10] of integer inv-squares =
      ( 1, 0.25, 0.1111, ...

3. function weightedSum(array[1..10] of integer a)
4. begin
5.    result = 0;
6.     for i = 1 to 10 do
7.         result := result + squares(i) * a(i);
8. end
```

On line 7, replace $+$ with $-$. Find the failure region, call it F_{AOR}. Next, replace the reference to the *squares* array with a reference to *inv-squares.* Find the failure region, call it F_{AAR}. Select an arbitrary point p in F_{AOR}. Does $p \in F_{AAR}$? With a probability very close to 1, it does. This scenario is quite common for all AAR mutants; they are coupled to other mutants and there seems to be little use of this mutation operator.

To summarize, this is a list of desirable properties of a *set* of mutation operators:

- Highly coupled to any mutant P_i (as defined in the previous section). Recognizing that these mutants must use the same language constructs

[6]Well, almost correct anyway—a future development would be to introduce some reliability measure of mutation testing. Maybe based on the coupling effect ratio?

as our set of mutation operators, it should be feasible to obtain high coupling. The exception seems to be when narrowing the input domain with for example if-statements; then a considerable part of the failure region is lost and so is the coupling effect ratio.

Example: If `if (x > 30 && x < 50) doSomething()` is missing, it could bring the coupling effect ratio down to a fraction of what is required to detect the bug.

- As a consequence of this, we need mutation operators that couples to constructs such as relational operators, function calls, array references, etc. A logical starting-point for constructing the set of operators would of course be to mutate these kinds of constructs. Ideally, they should cover the entire input domain:

$$\bigcup_i F_{P,\mu_i} = D \tag{3.10}$$

- Another consequence: the semantic size $s(P, \mu_i)$ for each mutation operator should be as small as possible. The one case guaranteed to detect all bugs would be the set of mutants covering the entire input domain D, where each mutant is killed by one specific value of the input domain.

- Low coupling with respect to one another. This will eliminate some operators. In fact, [OL93] showed empirically that test sets killing mutants generated by the five operators ABS, AOR, LCR, ROR and UOI was almost 100 % mutation adequate compared to the set of 22 operators on page 6.

3.4.2 What does Mutation Adequacy Measure?

Our primary goal using mutation testing is to deem a program well tested (or not). What does this mean? We already know that it does mutation adequacy does not imply a reliable test set (see equation 2.1). Assume we have 100 % mutation adequacy. What conclusions may we draw about the reliability of our program under test?

Consider the PLUS function below:

```
function plus(x, y: int): int
begin
    plus := x + y;
end
```

We have a clear vision what this function should do. It should add numbers. Period. Mutation testing allows us to make this assumption about a program, devise a testing strategy (write tests), and be fairly confident that the program under test does not exhibit any unexpected behaviour.

Chapter 4

Implementing a Mutation Testing System for Java and JUnit

One of the most widely used unit test frameworks is JUnit. It is designed to let programmers write tests in Java to test programs written in Java.

JUnit has TestCases and TestSuites. A TestSuite is a collection of TestCases and/or other TestSuites. A TestCase is the smallest testing unit and is supposed to test one single aspect of the program under test.

A mutation testing system for this framework is presented in the following sections. On a high level, these are the basic steps from source code to the completion of a test run:

1. Parse source code and create mutants, in this case a *metamutant*.
2. Find equivalent mutants.
3. Run test cases with unmutated program.
4. If passed, run test cases with mutants.

This process is shown in figure 4.1.

A complete, industrial strength tool would also include a repository for storing test runs, support for regression testing by running only mutants affected by code changes and possibly automatic test case generation.

Design goals:

- Should integrate well with existing tools.
- Should be easy to add new mutation operators.
- Should be language independent (to a certain degree, some elements like expressions are very similar across languages, whereas things like oo-features might not be).

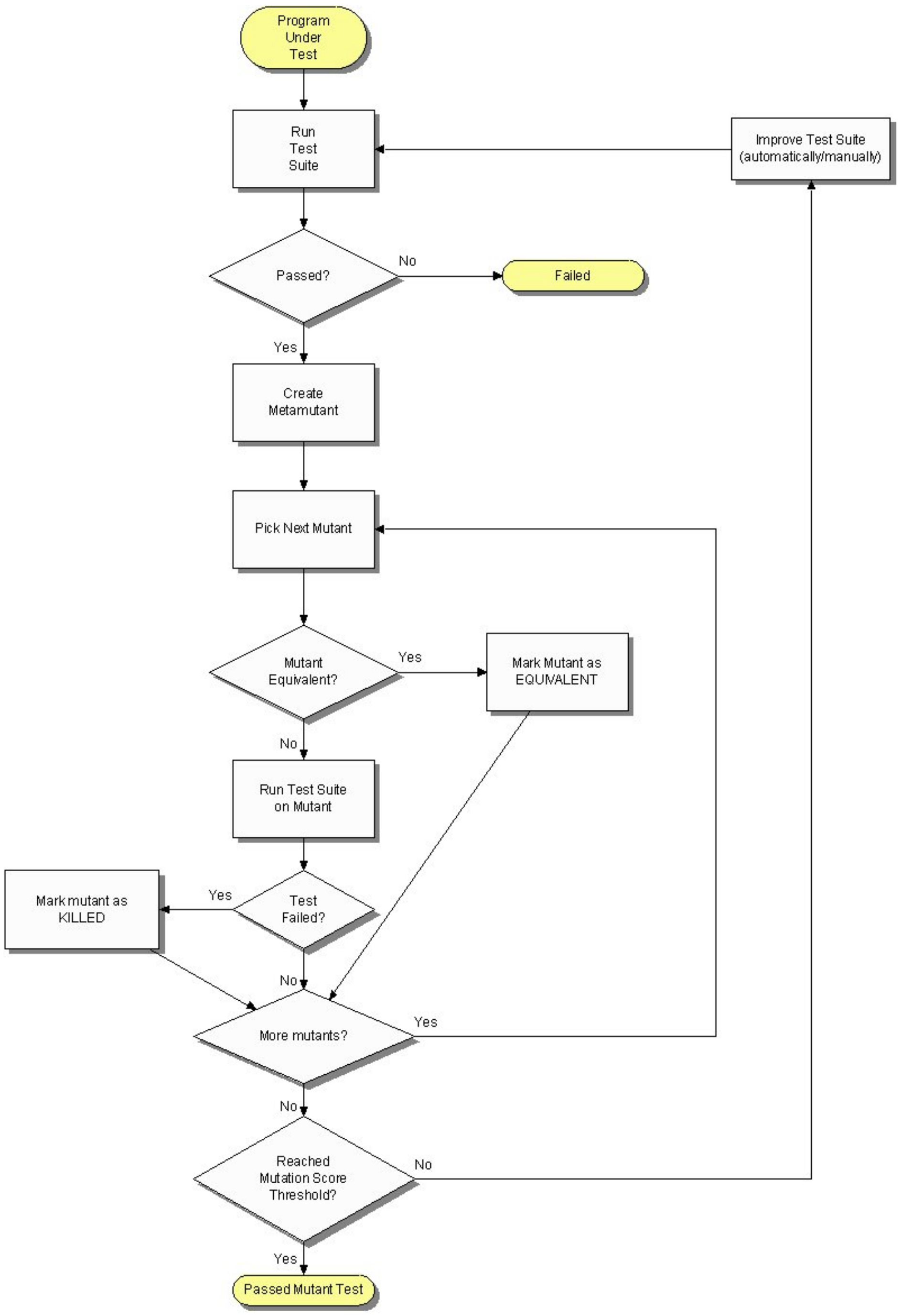

Figure 4.1: The mutation testing process implemented by the tool.

4.1 Metamutant Generator

The most important functionality of the program would of course be to create mutants. This section explains how to do that.

The problem is reduced to mutate individual program elements, since a mutant normally differs from the program under test in one program element only.

Consider this statement in the program under test:

```
...
z = x + y;
...
```

How do we mutate this statement? One approach is to create a metamutant [UN92]. A metamutant is one program containing all mutants. To declare which mutant is executing, an environment variable is set.

The metamutant version of the above statement could be something like

```
...
z = plusIntInt(x, y, 230, 232);
...
```

Each binary expression eligible for mutation is replaced with a function similar to the one above. The automatically generated `plusIntInt` function

```
...
plusIntInt(int x, int y, int firstMut, int lastMut)
{
    if (getCurrentMutation() >= firstmut &&
    getCurrentMutation() <= lastmut)
        {
            if (getCurrentMutation() == firstmut)
                return x - y;
            if (getCurrentMutation() == firstmut + 1)
                return x * y;
            if (getCurrentMutation() == firstmut + 2)
                return x / y;
            return x + y;
        }
        else
            return x + y;
}
...
```

checks whether, at this point in the program, it should execute a mutated statement. In the example, mutation number 230 mutates `x + y` into `x - y`, 231 into `x * y` and 232 into `x / y`. All other mutants executes the unmutated `x + y`.

4.1.1 Java Mutation Operators

Java is an object oriented programming language, very different from the procedural languages like FORTRAN and C which were the target languages for early mutation testing tools. Of course, this implies that the type of errors committed by programmers are different too. It is of great importance for a Java mutation testing tool to implement or allow for the implementation of mutation operators designed to uncover Java-specific faults. [MK02] proposes 24 new mutation operators listed in table 4.1 and [BI02] defines mutation operators for common Java classes like `Collection` and `InputStream`.

4.2 Equivalence Detector

Detecting equivalent mutants requires a constraint solver. Constraints are kept track of just like any scoped variable. Consider this code snippet:

```
1. public someFunc(int x, int y)
2. if (y == 0)
{
      ...
3.    z = plusIntInt(x, y, 230, 232);
      ...
}
```

A type checker knows that in line 3, the variables `x` and `y` are available. With not too much effort we can teach the type checker to handle constraints so that it also know that in the entire code block after line 2, the constraint `y = 0` holds (unless `y` is modified, of course).

For each node in the abstract syntax tree of the mutated program, the metamutant calls a method in a subclass of the `MutationOperator` class that either instruments the node or leaves it untouched. It is appropriate to let `MutationOperator` detect equivalent mutants. This code example demonstrates how this could be done for the `x + y` to `x - y` case:

```
public class ArithmeticOperatorMutationOp
...
public boolean isEquiv(BinaryExpression bexp, Constraints constr)
{
    ...
    if (bexp.getOperator() == PLUS &&
        mutateTo == MINUS)
    {
        Variable y = bexp.rightOperand();
        return constr.getConstraintsForVar(y).
```

Table 4.1: 24 Java mutation operators. The first letter of the mutation operator abbreviation represents the category; A = access control, I = inheritance, P = polymorphism, O = method overloading, J = Java-specific features, E = common programming mistakes.

Operator	Description
AMC	Access modifier change
IHD	Hiding varible deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	`super` keyword deletion
IPC	Explicit call of parent's constructor deletion
PNC	`new` method call with child class type
PMD	Instance variable deletion with parent class type
PPD	Paramter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloaded method contents change
OMD	Overloaded method deletion
OAO	Argument order change
OAN	Argument number change
JTD	`this` keyword deletion
JSC	`static` modifier change
JID	Member variable initialization deletion
JDC	Java supported default constructor create
EOA	Reference assignment and content assignment replacement
EOC	Reference comparison and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change

```
            equals(new EqualityConstraint(
                y, 0));
    }
    ...
}
```

The above code compares the constraints on `y`. If they are equivalent to the set of constraints { `y = 0` }, then the mutant is equivalent, otherwise it is not.

4.3 Test Runner

The architecture of JUnit lends itself easily to extraction of TestCases. The test runner runs the entire TestSuite against the unmutated program, records which tests execute which statements and, for each mutant, it runs all TestCases that executed the particular statement of the mutant. After the test run, every mutant is in one of the states *Not executed*, *Alive*, *Killed*, *Equivalent* or *Timed out* (if the execution of a test case did not terminate within a set time limit).

4.4 Testing the Triangle Program

The program in the appendix determines the type of a triangle; either it is illegal (the sides do not connect properly) or it is one of three valid cases scalene (no sides equal), isosceles (two sides equal) or equilateral (all sides equal).

The program is submitted to the mutation testing tool together with a pointer to the JUnit test class. Two experiments are to be performed. The first demonstrates the mutation adequacy of a test set known to be adequate from a testing perspective and the second uses the tool—or actually, a special API included with the tool—to investigate the coupling effect between mutants[1].

4.4.1 Equivalent Mutants

The problem with equivalent mutants still stands out as a time-consuming, error-prone and hence expensive task. To find all equivalent mutants in this specific case, we exhaustively tested every integer value in the domain $D = (x, y, z)$ where x, y and $z \in [-20, 40]$. Obviously, this method is infeasible in the general case.

[1]Only a subset of the available mutants are used in these experiments, including no special Java mutation operators.

Of the 117 mutants, these 9 (8 %) were found to be equivalent: 47, 81, 83, 96, 97, 99, 109, 110, 112. (Mutants 1 and 3 would be equivalent had the domain been limited to three integers; test case 13 tests with more and less parameters and kills those two mutants.) The numbers have no meaning other than identifying individual mutants.

4.4.2 Experiment 1: Mutation Adequacy of a Test Set Known to be Adequate

Myers [MY79] lists 13 test cases that thoroughly test the triangle program in the appendix:

1. A test case which represents a valid scalene triangle.
2. A test case which represents a valid equilateral triangle.
3. A test case which represents a valid isosceles triangle.
4. At least three test cases which represent valid isosceles. triangles such that you have tried all three permutations of two equal sides.
5. A test case in which one side is zero.
6. A test case in which one side is negative.
7. A test case with three positive integers such that the sum of two of them is equal to the third.
8. At least three test cases in category 7 such that you have tried all three permutations where the length of one side is equal to the sum of the lengths of the other two sides.
9. A test case with the sum of two of the numbers less than the third.
10. At least three cases in category 9 such that you have tried all three permutations.
11. A test case with all side lengths equal to zero.
12. At least one test case specifying non-integer values.
13. At least one test case specifying the wrong number of values (two or four).

These test cases (except for number 12, which is not applicable) were implemented as a JUnit test suite. Table 4.2 presents the results of the test run. We cannot draw any statistically valid conclusions based on this test run due to the limited number of mutants and the low complexity of our program under test. Yet, it is instructive to consider two things: that our test set indeed seem to be mutation adequate, although it could be better still, and the correlation between statement coverage and mutation score. To kill a mutant, it must be reached. If it is reached, the statement of that mutant is covered. Therefore, statement coverage must necessarily be a worse measure of test set adequacy[2].

Table 4.2: The percentage of non-equivalent mutants killed by each test case. The total is the mutation score of the entire test set. The "statement coverage" column shows the percentage of executed statements of the program under test.

Testcase	Mutation score	Statement coverage
1	36%	40%
2	23%	52%
3	32%	48%
4	49%	64%
5	3%	16%
6	3%	16%
7	34%	40%
8	44%	40%
9	19%	40%
10	26%	40%
11	0%	16%
13	5%	8%
Total	88%	96%

4.4.3 Experiment 2: The Coupling Effect Ratio Between Mutants

[SS90, BU80, WO93, MW95] examines mutation testing with only a random subset of all mutant programs. This is called *mutant sampling* and seems to be much more cost-effective than running all mutants. [WO93], for example, gives an instance where a 10 % sample of the mutants were only 16 % less effective than the entire set of mutants. Why does this work? One could suspect that the coupling of mutants plays an important role. One experiment we could do with our tool would be to extract all "necessary" mutants of the triangle program. The set of necessary mutants is almost as efficient as the entire set of mutants. This simple algorithm shows how to extract them:

- Define the set of necessary mutants, N. The initial value of N equals the entire set of mutants, $\{\mu_1, \ldots, \mu_n\}$.

[2]Several authors discuss the value of mutation testing compared to other coverage criteria. See for instance [FW94] and [MW93].

Table 4.3: The number of necessary mutants given a coupling effect ratio threshold. All mutants more coupled than the threshold to any necessary mutant is not a necessary mutant itself. The mutation score is the result of running the same triangle program test suite but with the necessary mutants only. (And yes, the middle column should be filled with 23's.)

Threshold	Necessary Mutants	Mutation Score
0.95	23	48%
0.90	23	48%
0.80	23	48%
0.70	23	48%

- For each mutant μ_i, if $\mu_i \in N$, compute the coupling effect ratio with respect to every other mutant in N.
- If the value of the coupling effect ratio with respect to the mutant μ_j is greater than a preset threshold, remove μ_j from N.

We used random testing to estimate the coupling effect ratio. For a set of n randomly distributed input values, compare the output out_0 with the output of the mutants, out_i, out_j. Count the number of inputs resulting in $out_0 \neq out_i$ (call this n_i) and the number of inputs resulting in both $out_0 \neq out_i$ and $out_0 \neq out_j$ (called $n_{i,j}$). Then the coupling effect ratio,

$$CeR_P(\mu_i, \mu_j) \approx \frac{n_{i,j}}{n_i} \tag{4.1}$$

Table 4.3 lists some important results. Of our 108 non-equivalent mutants, 23 couples more than 95 % to all the other mutants, which explains the success of previous results on mutation sampling[3]. The lower mutation score for the necessary mutants is not surprising as we expect the remaining alive mutants to not couple to the killed ones. We believe that this score is more accurate than the score of the entire mutant set. Also, a cost-effective set of mutation operators produce a high ratio of necessary mutants.

[3]Note that we already use a highly efficient subset of all mutation operators. The results would probably be even better if we used all of them.

Chapter 5

Conclusions

Mutation testing has been around since the late 1970s but is rarely used outside academia. Executing a huge number of mutants and finding equivalent mutants has been too expensive for practical use.

Selective mutation, the use of metamutants, automatic equivalence detection and progress in automatic test generation algorithms have opened the door to lowering the cost of mutation testing with orders of magnitude [OU00].

A usable mutation testing system for the software industry is within reach. There are still obstacles to overcome; the equivalent mutant problem remains, although to a lesser degree (and might even be ignored as 100 % adequacy is not always necessary). Then there are less theoretical issues. The system should integrate well with software development processes and software like testing frameworks and version control tools.

If these problems are resolved, mutation testing holds the potential of a truly powerful tool.

5.1 Mutation Operators

The common approach to implement mutation operators is mimicking potential faults like "inadvertently using the wrong logical connector" or "not overriding a method".

Even though inserting semantic faults is a most natural, simple and powerful way to create efficient mutants, the coupling effect is a function of semantic faults, not syntactic. We have clarified the concepts of failure regions and how they explain the coupling effect. In many papers, especially those treating the extension of mutation testing to object oriented languages and integration testing, no thorough semantic analysis of mutation operators is done. This casts doubts on the selection of mutation operators and could lead to either too many mutants (expensive) or too few (inadequate).

The four points on page 14 highlights the properties of an efficient set of mutation operators.

5.1.1 The Mutation Testing Tool

Much effort has been put into the making of a mutation testing tool for Java, a prototype of an easily extendible tool with support for JUnit, custom mutation operators, support for adding more programming languages in the future and a special API for conducting mutation testing experiments.

A couple of experiments to demonstrate the use and usefulness of the tool were performed. One of the experiments showed how to extract the set of necessary mutants from a larger set and might be used to gain more insight into efficient sets of mutation operators.

5.2 Future Work

During the course of writing this thesis, a few areas stood out as especially important (in order of importance):

- A mutation testing tool that is fast, easy to use and suitable for today's software development environments is much wanted.
- Because of mutation testing not being widely used in industry (because of the obstacles mentioned earlier), there is a lack of mutation testing experience. It would be interesting to implement a mutation testing process for various types of software projects and report the outcome.
- What about those equivalent mutants? Could we find better methods to find them? We should be able to find all equivalent mutants and/or safely ignore the rest. Very little or no human detection is essential for a successful solution to this problem.
- Efficient mutation operators are the key to quality mutation testing. Are the operators currently used for object-oriented languages and integration testing good enough?
- Contrary to what was assumed in section 3.2, domain sizes vary. For small domain sizes, like in the statement `if (x - y > z) then blah blah`, where the combined domain size of `x`, `y` and `z` is reduced to 2 (true/false), we expect the coupling effect to be reduced too. At this point, more test cases will be needed to assure (almost) correctness of the program.
- Mutation testing is quite flexible — for example, specialized mutation operators have been proposed for some Java container classes [BI02] and for finding environmental bugs [SP90]. Could this be exploited somehow?

Bibliography

[BD80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton and Frederick G. Sayward: Theoretical and Empirical Studies on Using Program Mutation To Test The Functional Correctness of Programs, *Proceedings of the 7th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, p.220–233, January 28–30, 1980, Las Vegas, Nevada.

[BI02] Roger T. Alexander, James M. Bieman, Sudipto Ghosh, Bixia Ji: Mutation of Java Objects, *13th International Symposium on Software Reliability Engineering (ISSRE'02)*, November 12–15, pp. 341–351, 2002.

[BU80] Timothy A. Budd: *Mutation Analysis of Program Test Data*, PhD Thesis, Yale University, New Haven CT, 1980.

[DM96] M. E. Delamaro, J. C. Maldonado, A. P. Mathur: Integration Testing Using Interface Mutation, *Proceedings of the Seventh International Symposium of Software Reliability Engineering (ISSRE'96)*, White Plains, NY, pp. 112–121, 1996.

[FW94] P. Frankl, S. Weiss and C. Hu: *All-uses Versus Mutation Testing: An Experimental Comparison of Effectiveness*, Technical report PUCS-100-94, Department of Computer Science, Polytechnic University, Brooklyn, NY, February 1994.

[HH99] R. M. Hierons, M. Harman and S. Danicic: Using Program Slicing to Assist in the Detection of Equivalent Mutants, *The Journal of Software Testing, Verification, and Reliability*, 9 4, pp. 233–262, 1999.

[HO82] W. E. Howden: Weak mutation testing and completeness of test sets, *IEEE Trans. on Softw. Eng.*, 8(4):371–379, July 1982.

[HS90] William Hsu, Mehmet Sahinoglu and Eugene H. Spafford: *An Experimental Approach to Statistical Mutation-Based Testing*, Technical Report SERC-TR-63-P, Software Engineering Research Center, Purdue University, 1990.

[MK02] Yu-Seung Ma, Yong-Rae Kwon and Jeff Offutt: Inter-Class Mutation Operators for Java, *Thirteenth International Symposium on Software Reliability Engineering*, Annapolis, MD, November 2002.

[NI02] Dept. of Commerce's National Institute of Standards and Technology: *NIST Planning Report 02-3*, The Economic Impacts of Inadequate Infrastructure for Software Testing, 2002.

[OF92] A. Jefferson Offutt: Investigations of the Software Testing Coupling Effect, *ACM Transactions on Software Engineering Methodology*, 1(1):3–18, January 1992

[OH96] A. Jefferson Offutt and J. Huffman Hayes: A Semantic Model of Program Faults, *Proc. Int. Symp on Software Testing and Analysis (ISSTA'96)*, 1996.

[OC94] A. Jefferson Offutt and W. Michael Craft: Using Compiler Optimization Techniques to Detect Equivalent Mutants, *The Journal of Software Testing, Verification and Reliability*, 4(3):131–154, September 1994.

[OL93] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland Untch and Christian Zapf: An Experimental Determination of Sufficient Mutation Operators, *ACM Trans. on Software Engineering & Methodology*, Vol. 5, pp. 99–118, April 1996.

[OP97] A. Jefferson Offutt and Jie Pan: Automatically Detecting Equivalent Mutants and Infeasible Paths, *The Journal of Software Testing, Verification, and Reliability*, Vol 7, No. 3, pp. 165–192, September 1997.

[OU00] A. Jefferson Offutt and Roland H. Untch: Mutation 2000: Uniting the Orthogonal, *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pp. 45–55, San Jose, CA, October 2000.

[MY79] G. J. Myers. *The Art of Software Testing*, Wiley-Interscience, New York, 1979.

[MR01] T. Murnane, K. Reed: On the Effectiveness of Mutation Analysis as a Black Box Testing Technique, *13th Australian Software Engineering Conference (ASWEC'01) August 27–28, 2001*, Canberra, Australia p. 0012, 2001.

[MW95] Aditya P. Mathur and W. Eric Wong: Reducing the Cost of Mutation Testing: An Empirical Study, *Journal of Systems and Software*, 31(3):185–196, December 1995.

[MW93] Aditya P. Mathur and W. Eric Wong: *Comparing the Fault Detection Effectiveness of Mutation and Data Flow Testing: An empirical study*, Technical Report SERC-TR-146-P, Software Engineering Research Center, Purdue University, West Lafayette, Indiana, March 1993.

[SP90] Eugene H. Spafford: Extending Mutation Testing to Find Environmental Bugs, *Software Practice and Experience*, 20(2):181–189, February 1990.

[SQAT] Software QA and Testing Resource Center: *Software QA and Testing Frequently-Asked-Questions, Part 1*, http://www.softwareqatest.com/qatfaq1.html (visited January 2003).

[SS90] M. Ṣahinoğlu and E. H. Spafford: A Bayes Sequential Statistical Procedure for Approving Software Products, *Proceedings of the IFIP Conference on Approving Software Products (ASP-90) (W. Ehrenberger, ed.)*, oo, pp. 43–56, Elsevier/North Holland, New York, Sept. 1990.

[UN92] R. H. Untch: Mutation-based Software Testing Using Program Schemata, *Proceedings of the 30th annual Southeast regional conference*, pp. 285–291, Raleigh, North Carolina, 1992.

[WA00] K. S. How Tai Wah: A Theoretical Study of Fault Coupling, *Software Testing, Verification and Reliability*, 2000; 10:3-45.

[WK00] Martin R. Woodward, Zuhoor A. Al-Khanjari: Testability, Fault Size and the Domain-to-Range Ratio: An Eternal Triangle, *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[WM97] W. E. Wong, J. C. Maldonado, M. E. Delamaro, and S. Souza: Use of Proteum to Accelerate Mutation Testing in C Programs, *Proc. of the 3rd ISSAT International Conference on Reliability and Quality in Design*, pp. 254–258, Anaheim, California, March 1997.

[WO93] W. E Wong: *On Mutation and Data Flow*, PhD Thesis, Technical Report SERC-TR-149-P, Purdue University, December 1993.

Appendix: The TRIANGLE program (in Java)

```
public static int getType(int side1, int side2, int side3)
{
    return getType(new int[] { side1, side2, side3 });
}

public static int getType(int[] sides)
{
    if (sides.length != 3)
        return ILLEGAL_ARGUMENTS;

    if (sides[0] < 0 || sides[1] < 0 || sides[2] < 0)
        return ILLEGAL_ARGUMENTS;

    int triang = 0;
    if (sides[0] == sides[1])
        triang = triang + 1;
    if (sides[1] == sides[2])
        triang = triang + 2;
    if (sides[1] == sides[2])
        triang = triang + 3;

    if (triang == 0)
    {
        if (sides[0] + sides[1] < sides[2] ||
            sides[1] + sides[2] < sides[0] ||
            sides[0] + sides[2] < sides[1])
            return ILLEGAL;
        else
            return SCALENE;
    }
    if (triang > 3)
        return EQUILATERAL;
    else if (triang == 1 && sides[0] + sides[1] > sides[2])
        return ISOSCELES;
    else if (triang == 2 && sides[0] + sides[2] > sides[1])
        return ISOSCELES;
    else if (triang == 3 && sides[1] + sides[2] > sides[0])
        return ISOSCELES;

    return ILLEGAL;
}
```