MUTATION TESTING

Παυλάκος Ηλίας - 3160143 , Αλεξάκης Δημήτριος - 3160003

Ανάλυση της μεθόδου ελέγχου λογισμικού mutation testing στα πλαίσια του μάθηματος Επαλήθευση, επικύρωση και συντήρηση λογισμικού του Τμήματος Πληροφορικής – Οικονομικό Πανεπιστίμιο Αθηνών

Καθηγητής
Μαλεύρης Νικόλαος

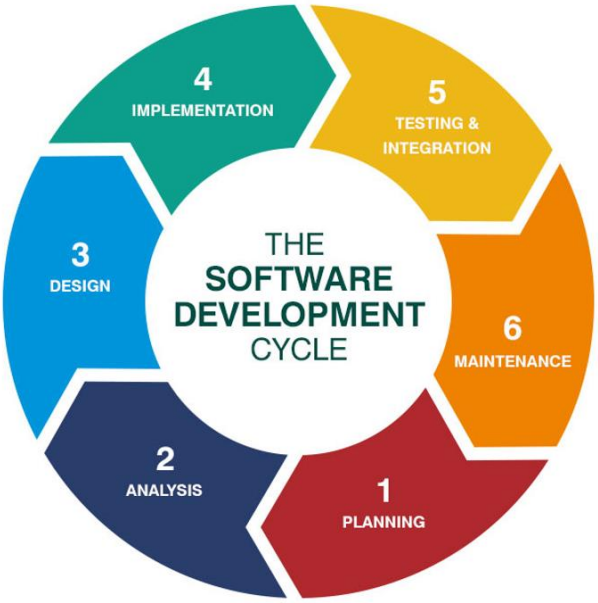
ΠΕΡΙΛΗΨΗ

Τα τελευτέα χρόνια έχουν υιοθετηθεί διάφορες μεθόδοι ελέγχου για την επικύρωση και αποσφαλμάτωση ενός συστήματος λογισμικού. Μια από αυτές ειναι η μέθοδος Mutation Testing (ή έλεγχος μεταλλάξεων) η οποία θεωρείται μια από τις πιο ισχυρές εκ των γνώστων μεθόδων ελέγχου αν όχι η ισχυρότερη. Η μέθοδος αυτή εμπεριέχει την παραμετροποίηση ενός προγράμματος με διάφορους μικρούς τρόπους. Κάθε νεό παραμετροποιήμενο πρόγραμμα που προκύπτει ονομάζεται mutant (ή μετάλλαξη). Μια σειρά απο tests βρίσκουν και απορρίπτουν mutants στους οποίους εντοπίστικαν οι διαφορές από το αρχικό πρόγραμμα. Άν ένας ή περισσότεροι mutants επιβιώσαν της παραπάνω διαδικασίας τότε είτε πρόκειται για κάποιον mutant που οι διαφορές του τον καθιστούν πανομοιότυπο με το αρχικό πρόγραμμα οπότε και απορρίπτεται, είτε έχει βρεθεί ένα πιθανό μέρος του προγράμματος που εμπεριέχει κάποιο κένο ασφαλείας. Αναπόφευκτα, το πόσο αποτελεσματική είναι αυτή η μέθοδος βασίζεται πολύ στο ποίοι mutants χρησιμοποιήθηκαν. Έτσι, εύκολα καταλαβαίνουμε ότι όσο μεγαλύτερος ο αριθμός mutants που θα παραχθούν και θα ελεγχθούν τόσο μεγαλύτερο κόστος και τόσο περισσότερος χρόνος χρειάζεται.

ΠΕΡΙΕΧΟΜΕΝΑ

ΕΙΣΑΓΩΓΗ

Ο έλεγχος ενός συστήματος λογισμικού είναι μια διαδικασία απαραίτητη για την ορθή και ομαλή ανάπτυξη ενός συστήματος λογισμικού. Τα τελευταία χρόνια με την όλο και ταχύτερη εξέλιξη της τεχνολογίας είναι αναπόφευκτο το λογισμικό να γίνεται όλο και πιο πολύπλοκο. Έτσι εμφανίστηκε η ανάγκη ανάπτυξης μεθόδων έλεγχου οι οποίοι είναι αξιόπιστοι αλλά και αποδοτικοί. Αξιόπιστοι ώστε να μπορεί ο δημιουργός να εγγυηθεί την ορθότητα του υπό ανάπτυξη λογισμικού. Αποδοτικοί ώστε στο περιορισμένο συνήθως χρόνο που έχει δοθεί στον έλεγχο του λογισμικού να μπορούμε να βγάλουμε επαρκή συμπεράσματα.



Μια τέτοια μέθοδος είναι και η Mutation Testing (ή έλεγχος μεταλλάξεων) η οποία είναι μια μέθοδος άσπρου κουτιού (ή White-box testing). Η τεχνική μετάλλαξης αποσκοπεί στην δημιουργία μεταλλαγμένων προγραμμάτων βασισμένα σε ένα αρχικό πηγαίο κώδικα. Κάθε μεταλλαγμένο πρόγραμμα (ή mutant) υφίσταται μικρές ατομικές αλλαγές ώστε να μην απέχει κατά πολύ από το αρχικό πρόγραμμα προς έλεγχο. Η φιλοσοφία πίσω από αυτό είναι ότι έλεγχοι οι οποίοι μπορούν να διακρίνουν την συμπεριφορά του αρχικού προγράμματος από αυτή των μεταλλάξεων, θα είναι σε θέση να διακρίνουν και τα λάθη ή πιθανά τμήματα νεκρού κώδικα (dead code) στο αρχικό πρόγραμμα υπό εξέταση. Γίνεται ήδη ξεκάθαρο ότι η αποτελεσματικότητα αυτής της μεθόδου εξαρτάται άμεσα από τις μεταλλάξεις που χρησιμοποιήθηκαν κατά την διάρκεια του έλεγχου.

Έτσι, χρειάζεται ιδιαίτερη προσοχή στην επιλογή των μεταλλάξεων καθώς πρόσφατες έρευνες έχουν δείξει ότι διαφορετικές ομάδες μεταλλάξεων μπορεί να οδηγήσουν σε διαφορετικά αποτελέσματα, γεγονός που υποδεικνύει και έναν από τους κίνδυνους της μεθόδου. Σε μεγάλα προγράμματα, όπως καταλαβαίνετε, ο όγκος των μεταλλάξεων θα είναι τεράστιος, καθώς και η υπολογιστική ισχύς που απαιτείται για να ελέγξουμε κάθε ένα από αυτά ξεχωριστά. Ωστόσο, μπορούμε πλέον με ασφάλεια να πούμε ότι οι περιπτώσεις εμφανίσεις περιορισμών λόγο της έλλειψης υπολογιστικής ισχύς είναι ελάχιστες στον τεχνολογικό κόσμο του σήμερα, ωστόσο κάτι τέτοιο δεν ισχύει και για τον χρόνο.

Οι απαιτήσεις ολοκλήρωσης λογισμικού σε στενά χρονικά περιθώρια όλο και αυξάνονται, έτσι πολύ ερευνητές έχουν ήδη ανάδειξη μεθόδους μειώσει του χρόνου επικύρωσης λογισμικού με τεχνικές ανάλυσης μεταλλάξεων. Πλέον, η θεωρία αλλά και η πράξη της μεθόδου έχουν ωριμάσει αρκετά για να μπορεί να χρησιμοποιηθεί σε εργαλεία ελέγχου που είναι ευρέος γνωστά όπως το muJava. Σε αυτήν την εργασία θα αναλύσουμε και θα κατανοήσουμε πως και γιατί δουλεύει αυτή η μέθοδος καθώς και πως μπορούμε να αποφύγουμε ή έστω να περιορίσουμε τους κίνδυνους που εμπεριέχει.

ΓΙΑΤΙ MUTATION TESTING?

Ο αυτοματοποιημένος έλεγχος έχει μπει για τα καλά στο λογισμικό μας, ειδικά τώρα που τα προγράμματα γίνονται όλο και πιο πολύπλοκα με χιλιάδες γραμμές κώδικα. Δημιουργούμε μια σειρά από ελέγχους (test suites) και βεβαιωνόμαστε ότι η συμπεριφορά του λογισμικού μας είναι η επιθυμητή τρέχοντας τους ελέγχους ξανά και ξανά. Έτσι, μια ομάδα θέλει να γνωρίζει αν πράγματι τα test που εκτελεί είναι καλά και αξιόπιστα, αν επαρκούν για τον πλήρη έλεγχο ή αν πρέπει να προσθέσει κι' άλλα.

Προκειμένου να απαντήσουμε σε αυτήν την ερώτηση χρησιμοποιούμε τον όρο κάλυψη κώδικα (ή code coverage) . Συνοπτικά, πρόκειται για το ποσοστό των γραμμών κώδικα που εκτελούνται κατά την διάρκεια μιας σειράς από ελέγχους. Άρα, εάν με τους ελέγχους μας καταφέρουμε να επιτύχουμε κάλυψη 100% τότε το λογισμικό μας δεν έχει κανένα λάθος και είναι έτοιμο για παράδοση στον πελάτη, σωστά? Η απάντηση είναι όχι. Η κάλυψη δεν είναι ένα μέτρο που μπορούμε να εμπιστευθούμε τυφλά και μπορεί πολλές φορές να μας δημιουργήσει την ψευδαίσθηση ότι το πρόγραμμα μας είναι αψεγάδιαστο. Μπορεί ήδη να έχετε καταλάβει γιατί ισχύει αυτό, αν όχι ας δούμε ένα παράδειγμα.

```
1 public boolean biggerThanTen(int x) {
2   if(x >= 10) {
3     return true;
4   } else {
5     return false;
6   }
7 }
```

Αυτή είναι μια απλή μέθοδος που επιστρέφει true μόνο αν το x είναι μεγαλύτερο ή ίσο του 10. Παρακάτω είναι ο έλεγχος που θα επιτελέσουμε:

```
1 @Test
2 public boolean myTest() {
3   assertTrue(biggerThanTen(11));
4   assertFalse(biggerThanTen(9));
5 }
```

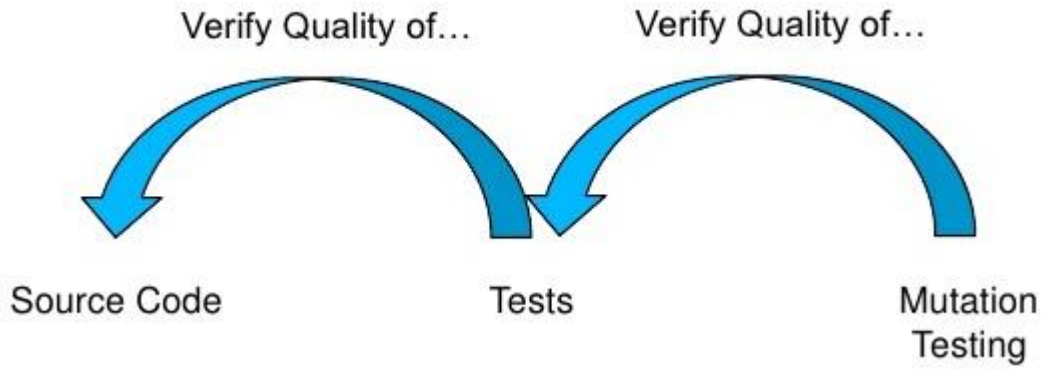
Μπορούμε εύκολα να συμπεράνουμε ότι η κάλυψη του ελέγχου είναι 100%.
Ωστόσο, δεν ελέγχουμε την ακραίες τιμές. Τι θα συμβεί αν δώσουμε στο X την τιμή 10. Είναι ξεκάθαρο πλέον ότι παρά το ποσοστό κάλυψης 100% που έχουμε πετύχει, ο έλεγχος μας έχει ένα πολύ μεγάλο λάθος και είναι ανεπαρκής.

Ας εξετάσουμε τον παρακάτω έλεγχο που διορθώνει τα λάθη του προηγούμενου και διατηρεί την κάλυψη σε ποσοστό 100%.

```
1 @Test
2 public boolean myTestV2() {
3   biggerThanTen(11);
4   biggerThanTen(10);
5   biggerThanTen(9);
6 }
```

Πράγματι, τώρα ο έλεγχος μας είναι πλήρης, ωστόσο έχουμε ξεχάσει να κάνουμε assert την τιμή επιστροφής της συνάρτησης. Ένα λάθος που όσο αστείο και αν φαίνεται στο παραπάνω παράδειγμα, σε μεγάλα συστήματα λογισμικού με χιλιάδες γραμμές κώδικα και πολλούς ανθρώπους να δουλεύουν στο ίδιο πρόγραμμα ίσως και να μην είχε βρεθεί και ποτέ χωρίς την χρήση κάποιου εξειδικευμένου εργαλείου.

Τώρα, που έχουμε καταλάβει ότι η κάλυψη δεν είναι επαρκής αυτούσια, πρέπει να βρούμε κάποια εναλλακτική. Εδώ είναι που η φιλοσοφία πίσω από τον έλεγχο μεταλλάξεων θριαμβεύει. Όχι μόνο ελέγχουμε το λογισμικό μας για λάθη στον πηγαίο κώδικα αλλά παράλληλα ελέγχουμε τους ίδιους τους ελέγχους. Εάν οι έλεγχοι μας είναι αδύναμοι παρά την κάλυψη που καταφέρνουν τότε δεν θα καταφέρουν να βρουν και να εξοντώσουν όλες τις μεταλλάξεις. Έτσι τα τελευταία χρόνια η μέθοδος mutation testing έχει αρχίσει να γίνεται από τις καλύτερες επιλογή στον έλεγχο λογισμικού.



ΙΣΤΟΡΙΚΗ ΑΝΑΔΡΟΜΗ

Αν αναλογιστούμε το γεγονος οτι η τεχνολογία, ιδιαίτερα στον κλάδο του προγραμματισμού βελτιώνεται διαρκώς, τότε εύλογα μπορεί να υποστηρίξει κάποιος ότι η τεχνική του mutation testing υπάρχει στο χώρο της ανάπτυξης λογισμικού εδώ και αρκετό καιρό. Η εν λόγω τεχνική, προτάθηκε αρχικά απο τον Richard Lipton στα χρόνια που υπήρξε μαθητης εν έτη 1971, και ύστερα η ιδέα διαμορφώθηκε και δημοσιοποιήθηκε από τον ίδιο, τον DeMillo και τον Sayward.

Η πρώτη υλοποίηση ενος εργαλείου ελέγχου μεταλλάξεων πραγματοποιήθηκε το 1980, απο τον Timothy Budd, στο Yale University ως μέρος του PhD του με το όνομα “Mutation Analysis”. Στα χρόνια που ακολούθησαν η επιστήμη των υπολογιστών ενστερνήστηκε και υιοθέτησε την έρευνα πάνω στην ανάλυση των μεταλλάξεων και μέσω της τεράστιας υπολογιστικής ισχύς που είναι διαθέσιμη στην κοινότητα η τεχνική και η ιδέα προσαρμόστηκε και αναπτύχθηκε άρτια στον κόσμο του προγραμματισμου. Με αυτόν τον τρόπο επιτεύχθηκε να καλφθεί μια τεράστια ανάγκη ανάπτυξης μεθόδων που υλοποιούν mutation testing σε αντικειμενοστρεφής γλώσσες προγραμματισμού, σε μη διαδικαστικες γλώσσες, αλλα και στην ανάπτυξη αυτομάτων μηχανών.

Το 2004 μια εταιρία, εν ονόματι Certess Inc. η οποία σήμερα είναι κομμάτι της Synopsys μετέφερε πολλές αρχές του mutation testing ακόμα και στον τομέα της επαλήθευσης υλισμικού. Στην ανάλυση μεταλλάξεων οποιαδήποτε αλλαγή επέρχεται στο λογισμικό, γινεται ορατη και εντοπίζεται μέσω της εξόδου του. Η Cerstess γι’ αυτόν τον λόγο ενσωμάτωσε ενα ελεγχτή ο οποίος πιστοποιεί το γεγονός οτι οποιαδήποτε αλλαγή θα ανιχνευθεί στην παραγώμενη έξοδο. Η διαδικασία αυτη ονομάστηκε functional qualification. [W]

ΥΠΟΘΕΣΕΙΣ MUTATION ANALYSIS

Αποτελεί αδιαμφισβήτητη αρχή, ότι το mutation testing είναι ένα πολύ χρήσιμο και ισχυρό εργαλείο στα χέρια οποιουδήποτε προγραμματιστή. Ωστόσο, όσο υποσχόμενες και αν φαίνονται οι τεχνικές που το πλαισιώνουν, είναι γεγονός ότι ο αριθμός των πιθανων λαθών που μπορεί να διακατέχουν ένα πρόγραμμα είναι τεράστιος. Με αυτή τη σκέψη, ορθολογικά μπορούμε να συμπαιράνουμε ότι όσο διεξοδικός και πολύπλευρος μπορεί να υπάρξει ο έλεγχος μεταλλάξεων, σε καμία περίπτωση δεν είναι σε θέση να εξαντλήσει την ύπαρξη όλων των πιθανών λαθών. Έτσι, το mutation testing στοχοποιεί κατα κόρων να εντοπίσει και να εξαντλήσει ένα υποσύνολο λαθών το οποίο τείνει να είναι πιο κοντά στη σωστή έκδοση του εκάστοτε προγράμματος, επιτυγχάνοντας με αυτόν τον τρόπο την θεωρητική εξάληψη όλων τον λαθών για ορθή λειτουργικότητα του λογισμικού υπό έλεγχο. Αυτή η θεωρία βασίζεται σε δύο υποθέσεις.

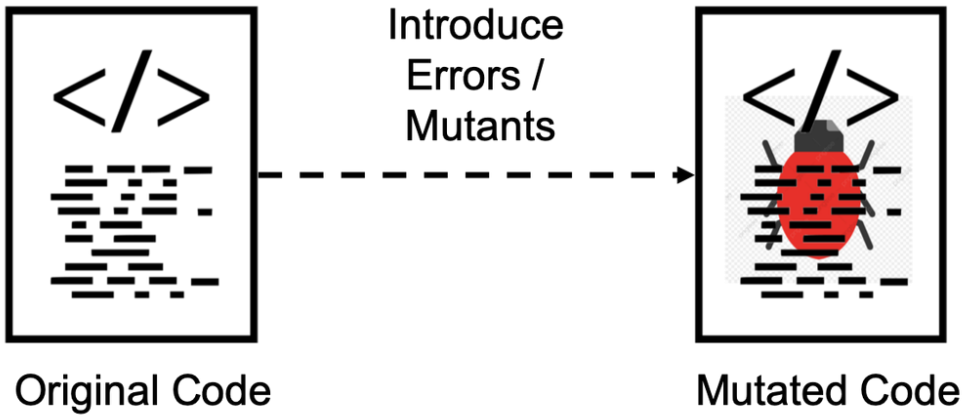
Η πρώτη υπόθεση ονομάζεται Competent Programmer Hypothesis (CPH) και η δεύτερη αναφέρεται ως Coupling Effect[W9]. Πιο συγκεκριμένα η Competent Programmer Hypothesis στηρίζεται στην άποψη ότι όλοι οι προγραμματιστές είναι ικανοί, κάτι το οποίο τους καθιστά κατάλληλους στο να σχεδιάζουν προγράμματα τα οποία τείνουν σε μεγάλο βαθμό στη σωστότερη τους έκδοση. Εύλογα άγεται το συμπέρασμα λοιπόν, ότι παρόλο που τα προγράμματα τα οποία υλοποιούνται απο ικανούς προγραμματιστές υπάρχει περίπτωση να περιέχουν λάθη, τα λάθη αυτά στη πλειοψηφία τους είναι σε θέση να διορθωθούν ύστερα από ορισμένες αλλαγές στο συντακτικό του προγράμματος. Αυτός είναι και ο λόγος που στον έλεγχο μεταλλάξεων προσομοιώνεται ένα σύνολο συντακτικών λαθών τα οποία θα μπορούσαν να έχουν υλοποιηθεί από έναν ικανό προγραμματιστή. [W10]

Η δεύτερη υπόθεση που θα εξετάσουμε αφορά το Coupling Effect η οποία προτάθηκε για πρώτη φορά από τον DeMillo το 1978. Η υπόθεση αυτή δεν επικεντρώνεται στην συμπεριφορά του πλήθους των προγραμματιστών, αλλα στον τύπο των λαθών που χρησιμοποιούνται για την ανάλυση των μεταλλάξεων. Πιο συγκεκριμένα, αυτό που πρεσβέυει η συγκεκριμένη υπόθεση είναι το γεγονός ότι τα δεδομένα των δοκιμών που χρησιμοποιούνται για την ανάλυση των μεταλλάξεων, τα οποία διαφέρουν από αυτά που θα μπορούσαν να θεωρηθούν σωστά, μόνο στην ύπαρξη ορισμένων απλών σφαλμάτων, είναι τόσο ευαίσθητα που δημιουργούν επιπρόσθετα, ακόμα πιο πολύπλοκα σφάλματα. Η υπόθεση αυτή επεκτάθηκε μέσω του Offutt ως την Coupling Effect Hypothesis η οποία και θεμελίωσε τον ορισμό του απλού και του σύνθετου σφάλματος και σύμφωνα με τον δικό του ορισμό ένα απλό σφάλμα αναπαριστάται από ένα απλό mutant, ενώ ένα σύνθετο σφάλμα αναπαριστάται από ένα σύνθετο mutant, το οποίο έχει δημιουργηθεί κάνοντας περισσότερες από μια αλλαγές.

THE MUTATION ANALYSIS PROCESS

Η ανάλυση μεταλλάξεων (ή Mutation Analysis) εισάγει λάθη μέσα στον πηγαίο κώδικα του προγράμματος προς έλεγχο δημιουργώντας πολλές εκδόσεις του του αρχικού προγράμματος όπου η κάθε μια περιέχει ένα λάθος. Ύστερα, μια σειρά από ελέγχους χρησιμοποιείται για να διαχωρίσουν τις λανθασμένες εκδόσεις από το αρχικό πρόγραμμα. Οι λανθασμένες εκδόσεις του αρχικού προγράμματος ονομάζονται mutants (ή μεταλλάξεις) ενώ η διαδικασία κατά την οποία βρίσκουμε τις διαφορές μιας μετάλλαξης και τις συγκρίνουμε με το αρχικό πρόγραμμα ονομάζεται killing the mutant (ή σκοτώνοντας την μετάλλαξη).

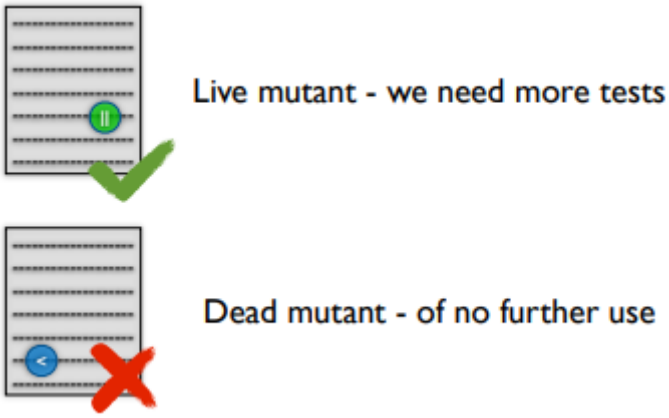
Οι μεταλλάξεις είτε αντιπροσωπεύουν πιθανά λάθη, τα οποία θα μπορούσε να είχε κάνει ο προγραμματιστής ή απαιτούν από το πρόγραμμα να εκτέλεση μια συγκεκριμένη ακολουθία εκτέλεσης. Μια τέτοια ακολουθία εκτέλεσης θα μπορούσε να αναφέρεται στον έλεγχο κάθε κλαδιού(branch) ενός προγράμματος ή να προκαλέσει όλες τις εξαιρέσεις σε ένα κομμάτι κώδικα. Οι μεταλλάξεις ωστόσο, είναι περιορισμένες σε μικρές και απλές αλλαγές βασισμένες στο Coupling Effect που αναλύσαμε στην προηγούμενη ενότητα.



Η ανάλυση μεταλλάξεων προσφέρει ένα κριτήριο ελέγχου παρά μια μέθοδο ελέγχου. Ένα κριτήριο ελέγχου (ή test criterion) είναι ένας κανόνας ή μια συλλογή από κανόνες που προσδίδει απαιτήσεις σε μια ομάδα έλεγχων. Οι δημιουργοί των ελέγχων μετρούν κατά πόσο ένα κριτήριο ικανοποιείται, η μέτρηση αυτή ονομάζεται κάλυψη. Για παράδειγμα, μια ομάδα από ελέγχους που πέτυχε κάλυψη 100% σημαίνει ότι ικανοποιεί πλήρως το κριτήριο που τέθηκε. Η κάλυψη μετριέται με βάση τις απαιτήσεις που έχουμε από την ομάδα ελέγχων που τέθηκε. Η μερική κάλυψη (ή partial coverage) μετριέται με βάση το ποσοστό των έλεγχων που ικανοποιούνται. Οι απαιτήσεις των ελέγχων είναι πολύ συγκεκριμένες και πρέπει να ικανοποιούνται ή

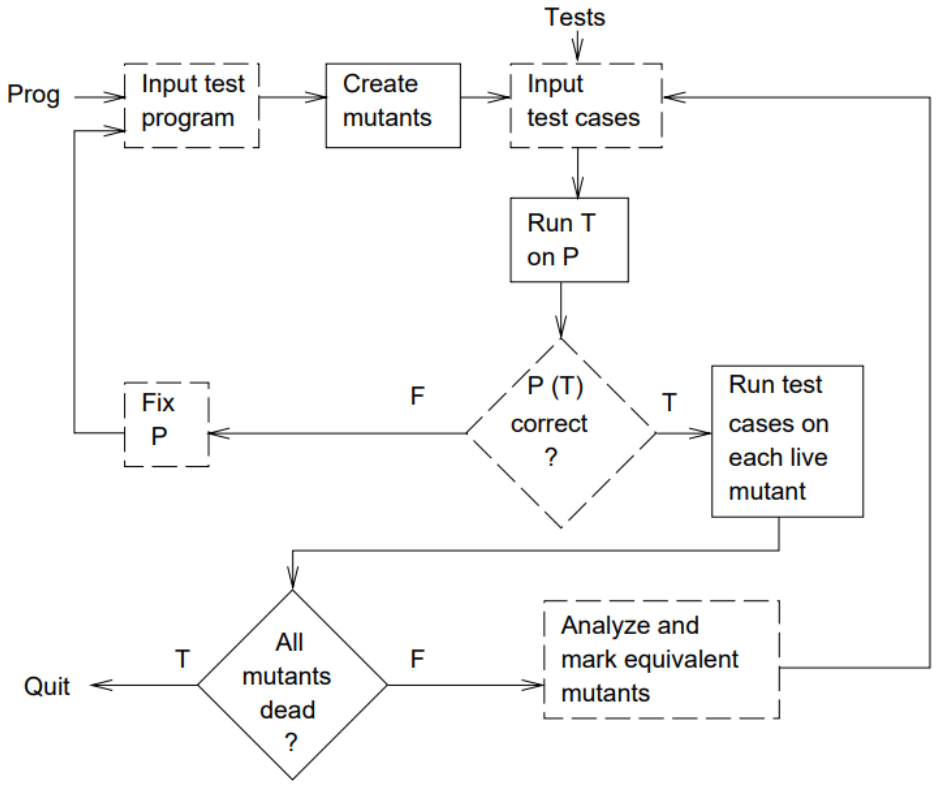
έστω να φτάνουν σε ένα αποδεκτό επίπεδο κάλυψης. Για παράδειγμα, η εκτελέσει όλων των διακλαδώσεων σε ένα πρόγραμμα είναι η απαίτηση για έλεγχο όλων τον τμημάτων του προγράμματος που σε διαφορετικές συνθήκες μπορεί να ήταν και dead-code, ή το σκότωμα των μεταλλάξεων είναι το κριτήριο του ελέγχου μεταλλάξεων. Έτσι, ένα κριτήριο ελέγχου καθορίζει το αν ο έλεγχος είναι αρκετός και καλύπτει τις απαιτήσεις.

Μια διαδικασία ελέγχου δίνει μια σειρά βημάτων που ακολουθάμε για να δημιουργήσουμε περιπτώσεις ελέγχου. Μπορεί να υπάρχουν πολλές διαδικασίες που χρησιμοποιούνται για να ικανοποιήσουμε ένα συγκεκριμένο κριτήριο αλλά μια διαδικασία ελέγχου δεν χρειάζεται να έχει ως στόχο της την ικανοποίηση κάποιου κριτηρίου. Συγκεκριμένα, η ανάλυση μεταλλάξεων είναι ένας τρόπος που μας επιτρέπει να μετράμε την ποιότητα τον περιπτώσεων ελέγχου και στην ουσία ο έλεγχος του προγράμματος είναι παραγόμενο της προηγούμενης διαδικασίας. Με απλά λόγια, στην πράξη το λογισμικό ελέγχεται, και μάλιστα με μεγάλη αξιοπιστία, ή οι περιπτώσεις ελέγχου δεν βρίσκουν τους mutants που σημαίνει πιθανό λάθος στο λογισμικό μας. Η παραπάνω ιδέα μπορεί να γίνει πιο εύκολα κατανοητή εξετάζοντας μια τυπική διαδικασία ανάλυσης μεταλλάξεων.



Όταν ένα πρόγραμμα τίθεται υπό την διαδικασία ανάλυσης μεταλλάξεων ενός συστήματος, αρχικά το σύστημα δημιουργεί μεταλλάξεις του προγράμματος. Ένας mutation operator (ή τελεστής μετάλλαξης) είναι ο κανόνας που διέπει την διαδικασία δημιουργίας μεταλλάξεων. Για παράδειγμα, τυπικά ένας mutation operator αντικαθιστά κάθε τελεστή με τον συντακτικά όμοιο του ανάλογα την γλώσσα προγραμματισμού, εισάγει καινούριους τελεστές, αλλάζει τελεστές με άλλους ανεξάρτητος την λειτουργία που επιτελούν ή ακόμα και διαγράφει ολόκληρες γραμμές κώδικα. Στο διάγραμμα 1 μπορούμε να δούμε γραφικά μία τυπική διαδικασία ανάλυσης μεταλλάξεων. Τα απλά ορθογώνια κουτιά αναπαριστούν τα

βήματα που γίνονταί αυτόματα κατά την διαδικασία του ελέγχου από το σύστημα. Ενώ τα κουτιά με διακεκομμένο περίγραμμα αναπαριστούν βήματα που πρέπει να γίνουν χειρωνακτικά.



Διάγραμμα 1: Τυπική διαδικασία ανάλυσης μεταλάξεων

Περιπτώσεις ελέγχου εισάγονται στο σύστημα προκείμενου να χρησιμοποιηθούν σαν είσοδοι στο πρόγραμμα προς έλεγχο. Αρχικά, κάθε περίπτωση ελέγχου εκτελείται στο αρχικό πρόγραμμα και ο άλεχτης εξετάζει την ορθότητα των αποτελεσμάτων. Εάν τα αποτελέσματα δεν είναι τα επιθυμητά, τότε ένα bug έχει βρεθεί και πρέπει να διορθωθεί πρώτου η συγκεκριμένη περίπτωση χρήσης χρησιμοποιηθεί ξανά Εάν τα αποτελέσματα ήταν σωστά, τότε η περίπτωση ελέγχου μπορεί να χρησιμοποιηθεί με ασφάλεια σε κάθε μετάλλαξη. Εάν η έξοδος μιας μετάλλαξης διαφέρει από αυτήν του αρχικού προγράμματος, τότε η μετάλλαξη χαρακτηρίζεται ως νεκρή. Οι νεκρές μεταλλάξεις δεν ελέγχονται από τις υπόλοιπες περιπτώσεις ελέγχου και αγνοούνται.

Όταν όλες οι περιπτώσεις ελέγχου έχουν εκτελεστεί, υπολογίζεται το mutation score (ή βαθμός μετάλλαξης). Πρόκειται για την αναλογία των νεκρών μεταλλάξεων έναντι του συνολικού αριθμού των μεταλλάξεων που δεν είναι πανομοιότυπες. Έτσι, σκοπός του ελενχτή είναι να καταφέρει να αυξήσει το mutation score στον βαθμό 1.00, κάτι που σημαίνει ότι όλες οι μεταλλάξεις έχουν εντοπιστεί επιτυχώς. Μια ομάδα από ελέγχους που καταφέρνει να σκοτώσει όλες τις μεταλλάξεις ονομάζεται επαρκής σε σχέση με τις μεταλλάξεις.

Εάν κάποιες μεταλλάξεις είναι ακόμα ζωντανές (δηλαδή δεν έχουν εντοπιστεί και σημειωθεί ως νεκρές), κάτι που είναι πολύ πιθανό να συμβαίνει, ο ελενχτής μπορεί να ενίσχυση τις περιπτώσεις ελέγχου βάζοντας καινούριες εισόδους. Υπάρχει η περίπτωση κάποιες μεταλλάξεις να είναι πανομοιότυπες με το αρχικό πρόγραμμα σημασιολογικά. Η έξοδος αυτών των μεταλλάξεων πάντα είναι πανομοιότυπη με αυτήν του αρχικού προγράμματος, έτσι δεν μπορούν να σκοτωθούν. Τέτοιες μεταλλάξεις δεν προσμετρούνται στον βαθμό μετάλλαξης.

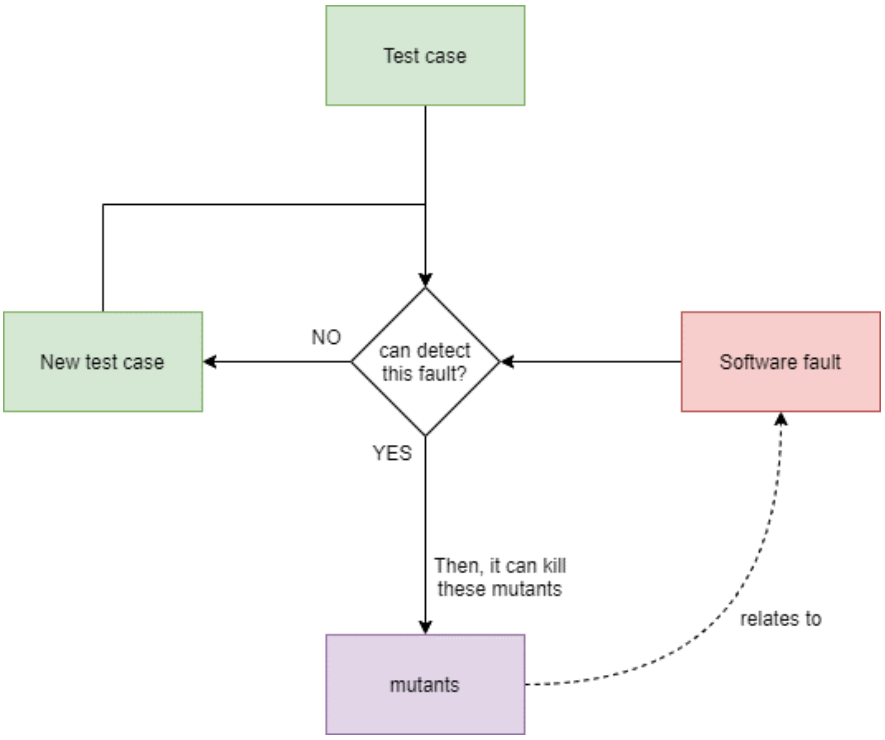
Εδώ αξίζει να σημειώσουμε ότι, ακόμα και αν ο ελενχτής δεν έχει βρει λάθη χρησιμοποιώντας τις προηγούμενες περιπτώσεις ελέγχου, ο βαθμός μετάλλαξης δίνει μια ένδειξη για την εξέλιξη του ελέγχου. Επόμενος, οι ζωντανές μεταλλάξεις υποδεικνύουν προς επαρκείς περιπτώσεις ελέγχου. Στις περισσότερες περιπτώσεις, ο ελενχτής δημιουργεί συγκεκριμένους ελέγχους προκειμένου να σκοτώσει συγκεκριμένες ζωντανές μεταλλάξεις. Αυτή η διαδικασία όπου ο ελενχτής προσθέτει καινούριες περιπτώσεις ελέγχου, επικυρώνει την ορθότητα των αποτελεσμάτων και σκοτώνει μεταλλάξεις επαναλαμβάνεται μέχρι ο ελενχτής να μείνει ικανοποιημένος με τον βαθμό μετάλλαξης. Ένα όριο στο βαθμό μετάλλαξης μπορεί να τεθεί προκειμένου οι ελένχτες να έχουν μια καλύτερη εικόνα της πορείας του ελέγχου μέχρι το αποδεκτό όριο.

MUTATION ANALYSIS ΣΤΟΝ ΕΝΤΟΠΙΣΜΟ ΣΦΑΛΜΑΤΩΝ

Σε αυτό το σημείο είναι εύκολο να σταματήσουμε την ανάπτυξη της θεωρίας και να επικεντρωθούμε στο πως θα χρησιμοποιήσουμε την ανάλυση μεταλλάξεων για να ελέγξουμε το πρόγραμμα μας. Ωστόσο υπάρχει ακόμα ένα μικρό κενό στην όλη ιδέα. Πως πραγματικά θα βρούμε τα σφάλματα του λογισμικού μας με την τεχνική ανάλυσης μεταλλάξεων?

Ένα σφάλμα λογισμικού είναι μια ομάδα από λάθος δηλώσεις στο πρόγραμμα που παρουσιάζει το σφάλμα. Τέτοια σφάλματα εντοπίζονται όταν ελέγχει εκτελούνται στο αρχικό πρόγραμμα μας και όχι σε κάποια μετάλλαξη. Αυτός που εκτελεί τους ελέγχους θα πρέπει να αποφασίσει αν η έξοδος του προγράμματος είναι η επιθυμητή. Εάν πράγματι είναι σωστή τότε η διαδικασία συνεχίζεται όπως περιγράφεται από πάνω. Διαφορετικά, αν η έξοδος δεν είναι η επιθυμητή, τότε ένα σφάλμα έχει βρεθεί και η διαδικασία στάματα μέχρι αυτό το σφάλμα να διορθωθεί και το πρόγραμμα μας να παράγει τα επιθυμητά αποτελέσματα.

Έτσι, καταλήγουμε στην θεμελιώδης αρχή του mutation testing στον έλεγχο σφαλμάτων όπως διατυπώθηκε από τον Geist Roberts[1]: Στην πράξη, αν το λογισμικό περιέχει ένα σφάλμα θα υπάρχει συνήθως μια ομάδα από μεταλλάξεις του αρχικού προγράμματος που μπορούν να εξοντωθούν μόνο από μια περίπτωση ελέγχου που εντοπίζει αυτό το σφάλμα.



ΠΑΡΑΔΕΙΓΜΑ ΕΦΑΡΜΟΓΗΣ MUTATION TESTING

Πλέον υπάρχουν πολλά mutation testing εργαλεία είτε open-source είτε διαθέσιμα στο εμπόριο. Συγκεκριμένα μερικά από τα πιο διάσημα είναι:

1. PIT
2. muJAVA
3. Jumble
4. Jester
5. Stryker
6. Many more….

Στο παρακάτω παράδειγμα θα χρησιμοποιήσουμε το Jumble προκειμένου να κάνουμε μια επίδειξη για το πως μπορεί να χρησιμοποιηθεί ένα τέτοιο εργαλεία για τον έλεγχο του λογισμικού μας. Το Jumble αλλάζει τον πηγαίο κώδικα σε επίπεδο byte, τρέχει μονάδες ελέγχου και δημιουργεί Mutations. Το Jumble είναι open-source και διαθέσιμο σαν add-on στο Eclipse.

Πριν εξετάσουμε το πρόγραμμα μας ας δούμε ένα παράδειγμα αλλαγής που επιτελεί το Jumble στον κωδικά μας σύμφωνα με τα πρότυπα που εξηγήσαμε στην ενότητα του mutation analysis.

Απλό κομμάτι κώδικα σε Java

```
int index = 0;
while(true)
{
   index++;
   if (index == 10)
        break;
}
```

Mutated κώδικας σε Java

```
int index = 0;
while (true)
{
   index++;
   if (index >= 10)
        break;
}
```

Όπως βλέπουμε, ο mutated κώδικας που παρουσιάσαμε παραπάνω θα περάσει το Jumble test επειδή η αλλαγή του τελεστή == σε >= δεν έχει κάποια επίπτωση στο αποτέλεσμα του κώδικα. Η εκτέλεση θα σταματούσε όταν η μεταβλητή index γινόταν ίση με 10 και εφόσον το index αυξάνεται κατά 1 και ξεκινάει με αρχική τιμή 0 το αποτέλεσμα του κώδικα παραμένει πανομοιότυπο.

Ας δούμε τώρα ένα πιο πολύπλοκο παράδειγμα. Ο κώδικας παρακάτω έχει γραφτεί και ελέχθη στο Eclipse με το Jumble plug in Το πρόγραμμα αυτό εντοπίζει την πρώτη εμφάνιση ενός διπλότυπου στοιχείου σε έναν πινάκα και επιστρέφει την τιμή του στοιχείου. Το πρόγραμμα περιέχει πολλά λάθη που θα προσπαθήσουμε να βρούμε με το Jumble.

```java
package testPackage;

import java.util.Arrays;
import java.util.List;
public class SampProg
{
    protected int repeatedNumber(final List a)
    {
        int len = a.size(),i,dup = -1;
        int[] arr = new int[len];
        for (i=0; i<len; i++)
        {
            arr[i] = a.get(i);
        }

        Arrays.sort(arr);
        try
        {
            for (i=1; i<len; i++)
            {
                if(arr[i] == arr[i-1])
                {
                    dup = arr[i];
                    break;
                }
            }
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
        }
        return dup;
    }
}
```

Αφού γράψουμε το πρόγραμμα, δημιουργούμε test cases χρησιμοποιώντας το JUnit σε java. Παρακάτω φαίνεται η έξοδος που παίρνουμε αφού εφαρμόσουμε Jumble Analysis. [V1]

```
Mutating testPackage.SampProg
Tests: testPackage.SampProgTest
Mutation points = 11, unit test time limit 2.94s
M FAIL: (testPackage.SampProg.java:8): -1 -> 1
M FAIL: (testPackage.SampProg.java:10): 0 -> 1
.M FAIL: (testPackage.SampProg.java:10): negated conditional
M FAIL: (testPackage.SampProg.java:16): 1 -> 0
M FAIL: (testPackage.SampProg.java:18): 1 -> 0
M FAIL: (testPackage.SampProg.java:18): - -> +
M FAIL: (testPackage.SampProg.java:18): negated conditional
M FAIL: (testPackage.SampProg.java:16): += -> -=
M FAIL: (testPackage.SampProg.java:16): negated conditional
.
Jumbling took 7.595s
Score: 18%
```

Ας εξετάσουμε την παραπάνω έξοδο:

1. Στην πρώτη γραμμή οπού εμφανίζεται η ετικέτα M FAIL η έξοδος -1 άλλαξε σε 1. Στο κύριο πρόγραμμα, εάν δεν έχουμε διπλότυπα περιμένουμε να επιστραφεί -1 αλλά αντί αυτού επιστρέφετε 1. Μπορούμε να διορθώσουμε την μετάλλαξη ενεργοποιώντας ελέγχους για τις τιμές.
2. Μετά, βλέπουμε “negated conditional” στην 3η γραμμή, όπως και στην 7η και 9η γραμμή. Εδώ δεν μπορούμε να διορθώσουμε το πρόβλημα.
3. Άλλες περιπτώσεις οπού η μετάλλαξη δεν κατάφερε να δώσει το αναμενόμενο αποτέλεσμα, είναι περιπτώσεις οπού τελεστές άλλαξαν σε άλλους τελεστές. Στην παραπάνω άλλα και σε άλλες περιπτώσεις, μεταλλάξεις που οφείλονται σε αλλαγές των τελεστών δίνουν μια καλή εικόνα στο που η περίπτωση ελέγχου είναι αδύναμη ή που υπάρχει σφάλμα στον κώδικα. Εδώ μπορούμε να βρούμε όλες τις μεταλλάξεις που εκτελεί το Jumble στον πηγαίο κώδικα.

ΠΡΟΒΛΗΜΑΤΑ MUTATION TESTING

Είναι πλέον κατανοητό σε όλους μας, οτι το mutation testing έχει έναν πολύ σημαντικό ρόλο στην ποιοτική ανάπτυξη και αναβάθμιση των λογισμικών ελέγχου προγραμμάτων. Παρόλα αυτά με την υλοποίηση του προκύπτουν αναπόφευκτα ορισμένα προβλήματα, τα οποία σε καμία περίπτωση δεν μπορούν να θεωρηθούν ως αμεληταία. Ένα από αυτά τα προβλήματα που δημιουργούνται είναι το γεγονός ότι προκειμένου να εφαρμόσουμε την τεχνική του mutation testing σε ένα set λογισμικού ελέγχου, θα χρειαστεί να χρησιμοποιήσουμε ένα πολυπληθή αριθμό από mutants γεγονός που απαιτεί πολύ μεγάλη υπολογιστική ισχύ για την εκτέλεσή τους.

Επιπρόσθετα, ένα ακόμα πρόβλημα που προκύπτει από την εφαρμογή της τεχνικής του mutation analysis είναι η αναπόφευκτη υποβολή ανθρώπινης προσπάθειας κατά την υλοποίησής της. Πιο συγκεκριμένα ο άνθρωπος είναι αναγκασμένος προκειμένου η τεχνική να είναι εφαρμόσιμη, να παρακολουθεί την έξοδο του αρχικού προγράμματος μετά το τέλος κάθε περίπτωσης ελέγχου. Είναι λογικά ορθή η άποψη οτι το συγκεκριμένο πρόβλημα δεν αποτελεί δυσκολία μόνο για την τεχνική του mutation analysis, αλλά αποτελεί γενικότερο πρόβλημα από τη στιγμή της ανάπτυξης της αλληλεπίδρασης του λογισμικού με το υλισμικο και τον άνθρωπο. Παρόλα αυτά, λόγο του τεραστίου αριθμού από mutants που απαιτεί κάθε υλοποίηση, καθιστούν την ανθρώπινη παρατήρηση και κατ' επέκταση την ύπαρξη ανθρώπινου δυναμικού, το πιο ακριβό κομμάτι του mutation testing. Ακόμα, λόγο της περιορισμένης ικανότητας εύρεσης ισοδύναμων mutant, πολλές φορές απαιτείται ακόμα περισσότερη προσπάθεια για την ανίχνευση της ισοδυναμίας των mutants ειδικα σε πολυπλοκα συστηματα λογισμικου.

Παρόλο που η πλήρης επίλυση των προαναφερθέντων προβλημάτων είναι αδύνατη, η ανάπτυξη και ή πολυετή έρευνα στον τομέα του της ανάλυσης μεταλλάξεων έχει οδηγήσει στην αυτοματοποίηση της τεχνικής σε λογικά πλαίσια και την ελλάτωση του συνολικού υπολογιστικού κόστους που προυποθέτει η συγκεκριμένη τεχνική.

ΤΕΧΝΙΚΕΣ ΕΛΑΧΙΣΤΟΠΟΙΗΣΗΣ ΤΟΥ ΚΟΣΤΟΥΣ

Αποτελεί ευρέως διαδεδομένη άποψη ακόμα και σήμερα, το γεγονός ότι το mutation testing είναι μια ακριβή τεχνική ελέγχου. Παρόλα αυτά η άποψη αυτή είναι εν μέρη ξεπερασμένη και βασίζεται στην ικασία ότι είναι αναγκαίο να συμπεριλάβουμε όλα τα διαθέσιμα mutants όπως δηλαδή εφαρμοζόταν η τεχνική στα πρώιμα της στάδια. Παρόλα αυτά, με την πάροδο του χρόνου και την συνεχή έρευνα και ανάπτυξη της τεχνικής το mutation testing έχει μετατραπεί σε μια πρακτική τεχνική ελέγχου, καθώς αρκετές τεχνικές μείωσης του κόστους έχουν εφαρμοστεί και έχουν προσαρμόσει την τεχνική στα δεδομένα της υπολογιστικής ισχύς που είναι διαθέσιμη. Οι τεχνικές μείωσης του κόστους που έχουν χρησιμοποιηθεί διακρίνονται σε τρεις κατηγορίες με τίτλους “κάνε λιγότερα”(do less), “κάνε πιο γρήγορα”(do faster) και ”κάνε πιο έξυπνα”(do smarter). Όσον αφορά την πρώτη κατηγορία οι τεχνικές που υπάρχουν ειναι οι Δηγματοληψία μεταλλάξεων(Mutant Sampling), Ομαδοποίηση μεταλλάξεων (Mutant Clustering) και η Επιλεκτική μεταλλάξη(Selective Mutation).

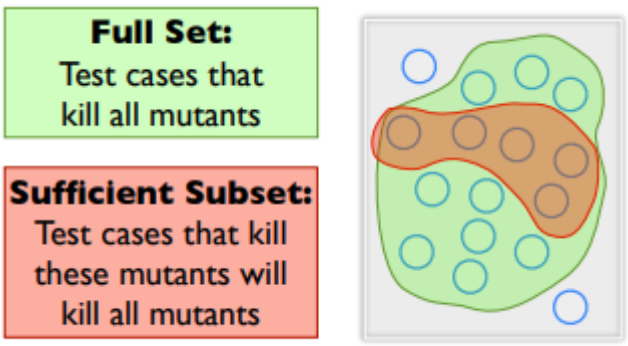
“DO LESS” APPROACHES

Η πρώτη από αυτές τις τεχνικές, δηλαδή η Mutation Sampling είναι μια απλή προσέγγιση στη μείωση του αριθμού των mutants, καθώς επιλέγουμε τυχαία ένα μικρό υποσύνολο από τον διαθέσιμο αριθμό των mutant. Η συγκεκριμένη ιδέα προτάθηκε για πρώτη φορά από τους Acree και Budd. Σε αυτή την προσέγγιση, αρχικά δημιουργούνται όλες οι δυνατές μορφές των μεταλλάξεων όπως συνέβαινε και στην πρώιμη μορφή του mutation testing. Μετά την δημιουργία των μεταλλάξεων μόνο ένα ποσοστό από αυτές επιλέγονται τυχαία και οι εναπομείναντες μεταλλάξεις αποσύρονται. Το σημαντικότερο κομμάτι αυτής της της τυχαίας επιλογής είναι το ποσοστό από το συνολικο set των mutant το οποίο θα χρησιμοποιηθεί. Στην έρευνα που διεξήγαγαν οι Wong και Mathur υπήρξε ένα πείραμα το οποίο υλοποιούσε τυχαίες επιλογές από mutants, ποσοστού ανάμεσα στο 10% εως και 40% του συνολικού αριθμού των mutants με διαδοχικά βήματα αύξησης του ποσοστού της ταξης του 5%. Το αποτέλεσμα αυτού του πειράματος έδειξε ότι η επιλογή του 10% σε σχέση με την επιλογή του συνολικού πλήθους των mutants είναι μόλις 16% λιγότερο αποδοτική όσον αφορά το mutation score. Η μελέτη αυτή απέδειξε ότι η επιλογή του πλήθους των μεταλλάξεων ποσοστού πάνω από το 10% του συνολικού πλήθους παρέχει έγκυρα αποτελέσματα. [2] [3] [4]

Μεταβαίνοντας στην δεύτερη τεχνική, δηλαδή αυτή της ομαδοποίησης μπορούμε εύκολα να διαπιστώσουμε ότι παρόλο που η ιδεολογία για τη μείωση τού κόστους και σε αυτή την περίπτωση πραγματεύεται την ελλάτωση του πλήθους των συνολικών mutants που επιλέγουμε, η μεθοδολογία που ακολουθείται είναι πολύ

διαφορετική. Αναλυτικότερα, η ιδέα για την ομαδοποίση των mutant διαφοροποιείται από αυτή της Δειγματοληψίας, διότι στη συγκεκριμένη περίπτωση το υποσύνολο των mutants που θα χρησιμοποιηθούν δεν επιλέγονται τυχαία, αλλά προκύπτουν μέσα από ειδικά σχεδιασμένους αλγορίθμους ομαδοποίησης. Αρχικά δημιουργούνται όλα τα mutant πρώτης τάξεως και στη συνέχεια ένας αλγόριθμος ομαδοποίσης διαχωρίζει σε ομάδες τα παραγμένα mutants με βάση την εν δυνάμει χρήση τους στη κατηγορία των σφαλμάτων τα οποία περιέχουν, με σκοπό να σκοτωθούν από το εκάστοτε τεστ. Τέλος για το mutation testing επιλέγεται μόνο ένα μικρό μέρος από το πλήθος της κάθε ομάδας, ενώ όσα mutants δεν επιλέγονται αποσύρονται και δεν χρησιμοποιούνται. Με αυτό τον τρόπο ο συνολικός αριθμός των mutants που χρησιμοποιείται μειώνεται αισθητά χωρίς όμως να υπονομεύεται η αξιοπιστία του mutation testing, καθώς το mutation score διατηρείται σε έγκυρο επίπεδο.

Όσον αφορά την τεχνική της Επιλεκτικής μεταλλάξης(Selective Mutation) ο τρόπος που επιτυγχάνεται η μείωση των mutants είναι πολύ διαφορετικός σε σχέση με τις προηγούμενες δύο τεχνικές. Η βασική ιδέα αρχικά προτάθηκε ως περιορισμένη μετάλλαξη από τον Mathur, ενώ ακολούθως διευρύνοντας την ιδέα ονομάστηκε επιλεκτική μετάλλαξη απο τον Offutt. Ο τρόπος λειτουργίας για τη συγκεκριμένη τεχνική είναι να μειώσουμε το πλήθος των τελεστών που χρησιμοποιούμε για να δημιουργήσουμε τις μεταλλάξεις και όχι τις μεταλλάξεις καθ' αυτές. Πιο συγκεκριμένα η διαδικασία που ακολουθείται ξεκινάει με την επιλογή ενός υποσυνόλου των συνολικών τελεστών μετάλλαξης από τους οποίους και θα παραχθούν όλα τα δυνατά mutants των οποίων ο πληθικός αριθμός θα είναι με τη σειρά του κι αυτός υποσύνολο των συνολικών mutants που θα είχαμε στη διάθεση μας, αν χρησιμοποιούσαμε όλους τους τελεστές χωρίς όμως να ελλατώσουμε την αξιοπιστία του ελέγχου. Είναι γεγονός ότι διαφορετικοί τελεστές μετάλλαξης δημιουργούν διαφορετικό πλήθος από mutants και κάποιοι από αυτούς δημιουργούν σημαντικά μεγαλύτερο αριθμό mutants από άλλους όπου ορισμένες μεταλλάξεις από τις παραγώμενες ίσως θεωρηθούν περιττές. Για παράδειγμα οι τελεστές μετάλλαξης ASR και SVR είναι δύο από τους 22 Mothra τελεστές οι οποί είναι υπεύθυνοι για τη δημιουργία των 30%-40% των συνολικών mutants. Προκειμένου να μειωθεί ο συνολικός αριθμός των παραγώμενων μεταλλάξεων, ο



Mathur πρότεινε να παραληφθούν οι δύο αυτοί τελεστές. Η πρόταση αυτή υλοποιήθηκε από τον Offutt, ο οποίος στη συνέχεια έκανε πειράματα που υλοποιούσαν την απόρριψη τεσσάρων και έξι τελεστών μετάλλαξης αντίστοιχα. Στην έρευνα των Mathur και Offutt υπήρξε η παρατήρηση ότι η παράληψη δυο τελεστών μετάλλαξης μπορεί να επιφέρει mutation score ίσο με 99.99% με μείωση του συνολικού αριθμού των mutant της ταξης του 24%, ενώ η παράληψη τεσσάρων τελεστών επέφερε mutation score 99.84% και μείωση των mutants κατά 41%.Τέλος, απορρίπτοντας 6 τελεστές μετάλλαξης το mutation score μεταβλήθηκε στο 88.71% και ο αριθμος των συνολικών των mutants μειώθηκε κατά 60%. [5] [6] [7]

"DO FASTER" APPROACHES

Τα περισσότερα mutation testing συστήματα λογισμικού λειτουργούν ερμηνεύοντας πολλές μικρές διαφορετικές εκδόσεις του ίδιου προγράμματος. Αν και τέτοιου είδους συστήματα κάνουν την διαχείριση τον μεταλλάξεων πιο βολική, αυτή η ευκολία έρχεται με τα μειονεκτήματα της. Αυτόματα συστήματα ανάλυσης μεταλλάξεων βασισμένα στις κλασικές μεθόδους ερμηνείας μεταλλάξεων είναι αργά, εξαιρετικά πολύπλοκα για να φτιαχτούν και συνήθως δεν είναι σε θέση να προσομοιώσουν με ακρίβεια της πραγματικές συνθήκες που πρόκειται να χρησιμοποιηθεί το λογισμικό. Για να λύσουμε αυτό το πρόβλημα, ο ερευνητής R. Untch δημιουργισε μια νέα μεθοδο, το Mutant Schema Generation(MSG). [8] [9]

Αντί να μεταλλάσσουμε το αρχικό πρόγραμμα και να διερμηνεύουμε έλεγχους στις μεταλλάξεις μια προς μια όπως οι κλασικές μέθοδοι, τώρα κωδικοποιούμε όλες τις μεταλλάξεις σε ένα source-level πρόγραμμα που ονομάζεται 'metamutant'. Αυτό το πρόγραμμα μετά γίνεται compile με τον ίδιο compiler που χρησιμοποιείται στην ανάπτυξη του αρχικού προγράμματος και εκτελείται στο ίδιο περιβάλλον που εκτελείται το αρχικό πρόγραμμα. Επειδή μέθοδοι mutation testing που στηρίζονται που στηρίζονται σε αυτήν την τεχνική δεν χρειάζεται να παρέχουν ολοκληρωμένα runtime semantics και διαφορετικά περιβάλλοντα εκτέλεσης, είναι πιο άπλα και ευκολότερα στο να φτιαχτούν, καθώς και πιο φορητά Δοκιμές έδειξαν ότι το σύστημα

TUMS που ακολουθεί το MSG μοντέλο, είναι σημαντικά γρηγορότερα από το Mortha, με ταχύτητες ακόμα και διπλάσιες κάποιες φορές.

Άλλη μια μέθοδος για να αποφύγουμε τον κλασικό τρόπο εκτέλεσης είναι η εκτέλεση κάθε mutant ξεχωριστά οπού κάθε mutant φτιάχνετε, γίνεται compile, συνδέεται και εκτελείται ξεχωριστά. Το σύστημα Proteum είναι ένα παράδειγμα τέτοιας μεθόδου. Ένα τέτοιο σύστημα μπορεί να πετύχει ακόμα και ταχύτητες 15 με 20 φορές μεγάλυνες από τα παραδοσιακά όταν η ταχύτητα εκτέλεσης κάθε mutant υπερβαίνει κατά πολύ το compile και την σύνδεση. Ωστόσο, πρόκειται για μέθοδο high risk high reward αφού σε περίπτωση που αυτή η συνθήκη δεν ισχύει οι ταχύτητες μπορεί να είναι πολύ πιο αργές από μια παραδοσιακή μέθοδο.

Για να αποφύγουμε τέτοιου είδους ρίσκα , οι DeMillo, Krauser, και Mathur ανέπτυξαν ένα πρόγραμμα ενσωματωμένο στον compiler που αποφεύγει την συμφόρηση άλλα εκτελεί κανονικά τον compiled κωδικά Σε αυτή την μέθοδο, το πρόγραμμα υπό έλεγχο γίνεται compile με έναν ειδικό compiler. Όσο η διαδικασία του compile εξελίσσεται, οι ιδιότητες τον μεταλλάξεων σημειώνονται και κομμάτια κωδικά που παρουσιάζουν συνοπτικά αυτές τις μεταλλάξεις ετοιμάζονται. Η εκτέλεση μιας συγκεκριμένης μετάλλαξης απαιτεί μονό το συγκεκριμένο κομμάτι κώδικα να ενσωματωθεί πριν την εκτέλεση. Αυτή η ενσωμάτωση είναι φτηνή και η μετάλλαξη εκτελείται σε ταχύτητες compile.

“DO SMARTER” APPROACHES

Συστήματα λογισμικού όπως η Mortha εκτελούν τις μεταλλάξεις μέχρις ότου τελειώσουν, ύστερα συγκρίνουν τα αποτελέσματα τους με αυτά του αρχικού προγράμματος υπό έλεγχο. Weak mutation (ή μέθοδος ασθενών μεταλλάξεων) που αρχικά προτάθηκε από την ερευνητή S. Howden, είναι μια προσεγγιστική τεχνική που συγκρίνει την εσωτερική κατάσταση των μεταλλάξεων και του αρχικού προγράμματος αμέσως μετά την εκτελέσει του μεταλλαγμένου προγράμματος.

Το συστήματα λογισμικού Leonardo[8], που υλοποιήθηκε ως μέρος του Mortha έκανε δυο πράγματα. Πρώτον, υλοποίησε ένα λειτουργικό σύστημα έλεγχου ασθενών μεταλλάξεων τα αποτελέσματα του οποίου θα μπορούσαν εύκολα να συγκριθούν σε 4 διαφορετικές στιγμές με τις παραδοσιακές μεθόδους έλεγχου μεταλλάξεων. (1) Μετά την πρώτη διερμηνεία της δήλωσης που εμπεριέχει το mutated symbol. (2) Μετά την πρώτη εκτέλεση της μεταλλαγμένης δήλωσης. (3) Μετά την πρώτη εκτέλεση του κόμματοι κωδικά που εμπεριέχει την μεταλλαγμένη δήλωση (4) Μετά από καθε εκτέλεση του βασικού κομματιού κωδικά που περιέχει την μεταλλαγμένη δήλωση Η χρήση του συστήματος Leonardo έδειξε ότι η μέθοδος ασθενών μεταλλάξεων μπορεί να παράξει ελέγχους που είναι όσο αποτελεσματικοί όσο έλεγχοι που παράγονται από τις κλασικές μεθόδους strong mutatuion και ο

χρόνος εκτέλεσης μειωνόταν κατά 50% ή περισσότερο κάποιες φορές. Ακόμη, βρέθηκε ότι η πιο αποτελεσματική στιγμή σύγκρισης τον μεταλλάξεων ήταν μετά την πρώτη εκτέλεση την μεταλλαγμένης δήλωσης.

Η χρήση κατανεμημένων συστημάτων για τον διαμοιρασμό του υπολογιστικού φόρτου αποτελεί μια άλλη “do smarter” approach. Πολλοί ερευνητές έχουν προσπαθήσει να ενσωματώσουν συστήματα ανάλυσης μεταλλάξεων σε συστήματα όπως SIMD machines, Hypercube (MIMD) και network(MIMD) συστήματα. Επειδή κάθε μετάλλαξη είναι ανεξάρτητη των υπόλοιπων, το κόστος επικοινωνίας παραμένει αρκετά χαμηλά Ένα από τα εργαλεία που αναπτυχθήκαν κατάφερε να επιτύχει ομαλή ταχύτητα για μέτριου μεγέθους συνάρτησης πηγαίου κωδικά. [10] [12]

Ο Fleyshgakker και Weiss ανέπτυξαν αλγόριθμους που βελτιώνουν την ταυτότητα εκτέλεσης των παραδοσιακών συστημάτων mutation analysis με το μειονέκτημα της χρήσης μεγαλύτερου αποθηκευτικού χώρου. Με την έξυπνη αποθήκευση πληροφοριών για τις καταστάσεων των μεταλλάξεων, η τεχνική τους βοηθάει σημαντικά στην διάκριση για το ποιες μεταλλάξεις είναι έτοιμες προς εκτέλεση. [11]

SECOND ORDER MUTATION TESTING

Μια άλλη μέθοδος που προτάθηκε για να μειωθεί η διαδικασία του ελέγχου των μεταλλάξεων είναι βασισμένη σε μεταλλάξεις που φέρουν έναν αριθμό προτεραιότητας. Σύμφωνα με αυτή την μέθοδο, δοθέντων μια ομάδας από μεταλλάξεις των παραδοσιακών μεθόδων (first-order), παράγεται μια νέα ομάδα με λιγότερες μεταλλάξεις. Κάθε μετάλλαξη σε αυτή την νέα ομάδα θα έχει δυο λάθη ταυτόχρονα μέσα στον πηγαίο κώδικα της και κάθε first-order μετάλλαξη θα περιέχετε σε τουλάχιστον μια second-order μετάλλαξη. Έτσι, ο αριθμός τον τελικών μεταλλάξεων προς έλεγχο είναι σχεδόν ο μίσος των αρχικών first-order μεταλλάξεων.

Διαφορετικές στρατηγικές για το πως οι first-order μεταλλάξεις πρέπει να συνδυαστούν έχουν προταθεί. Ο ερευνητής Polo Et έχει προτείνει τις ακόλουθες στρατηγικές: RandomMix , DifferentOperators , LastToFirst. Στην πρώτη στρατηγική, ο συνδυασμός γίνεται από τυχαία διαλεγμένα ζευγάρια από first-order μεταλλάξεις χρησιμοποιώντας κάθε μετάλλαξη μια φορά. Στην δεύτερη, κάθε συνδυασμός μεταλλάξεων χρησιμοποιεί μεταλλάξεις που έχουν διαφορετικούς τελεστές μεταλλάξεις. Στην τελευταία στρατηγική, οι μεταλλάξεις συνδικάζονται με βάση την σειρά που παραχθήκαν από το εργαλείο μετάλλαξης που χρησιμοποιείται. Η πρώτη μετάλλαξη συνδυάζεται με την τελευταία, η δεύτερη με την προτελευταία και ούτω καθεξής.

Τα αποτελέσματα που προκύπτουν εφαρμόζοντας τις παραπάνω στρατηγικές είναι πολύ ενθαρρυντικά, μιας και παρατηρήθηκε μεγάλη μείωση στον αριθμό τον ομοιών μεταλλάξεων. Ωστόσο, παρατηρήθηκε και μια μικρή μείωση στην ποιότητα τον ελέγχων με βάση τις second-order μεταλλάξεις. Ωστόσο, τα αποτελέσματα της ερευνάς αφορούσαν μικρά προγράμματα και χρειάζεται παραπάνω ερευνά για να διαπιστωθεί η αποτελεσματικότητα της μεθόδου.

Σύμφωνα με έρευνα [13] στην αποτελεσματικότητα των μεθόδων των ερευνητών Νικόλαος Μαλεύρης και Μιχάλης Παπαδάκης, οι μέθοδοι first-order είναι γενικά πιο αποτελεσματικές στην ανίχνευση σφαλμάτων από τις second-order αλλά με μεγαλύτερο κόστος. Οι second-order μέθοδοι μπορούν να μειώσουν δραστικά των αριθμό των μεταλλάξεων με αποτέλεσμα να μειώνουν και τον αριθμό των απαιτούμενων ελέγχων. Σύμφωνα με τα αποτελέσματα της έρευνας, οι όμοιες μεταλλάξεις μπορούν να μειωθούν κατά 80% με 90% με την χρήση second-order mutation τεχνικών. Ακόμη, οι second-order τεχνικές μπορούν να επιτύχουν μειώσει 30% στους απαιτούμενους ελέγχους με σχεδόν 10% μειώσει στην ικανότητα τους να βρίσκουν σφάλματα σε σχέση με τις first-order τεχνικές.

ΕΠΙΛΟΓΟΣ

Η μέθοδος ελέγχου λογισμικού mutation testing είναι μια ακόμη εξελισσόμενη μέθοδος που μπορεί να φέρει τεράστια πλεονεκτήματα στην ανάπτυξη λογισμικού. Με τους όλο και αυξανομένους υπολογιστικούς πόρους, μπορεί να διεισδύσει βαθύτερα μέσα στην βιομηχανία ανάπτυξης λογισμικού και να βοηθήσει στην δημιουργία πιο ποιοτικού λογισμικού. Ωστόσο, υπάρχουν ακόμα πλευρές τις μεθόδου που καθυστερούν την ενσωμάτωση της στην βιομηχανία. Το κύριο μειονέκτημα της είναι η υπολογιστική πολυπλοκότητα που εισάγει, οι ισοδύναμες μεταλλάξεις και η έλλειψη εργαλείων ενσωμάτωσης της είναι αλλά προβλήματα που σιγουρά θα λυθούν σύντομα.

Βέβαια, με την όλο και πιο γρήγορη πρόοδο στον τομέα τις AI και του machine learning και οι εφαρμογές που θα μπορούσαν να έχουν τέτοιες τεχνολογίες στην ανάλυση μεταλλάξεων, όπως η μείωση της υπολογιστικής πολυπλοκότητας και η αυτοματοποίηση της μεθόδου, θα μπορούσαν να φέρουν την επανάσταση στον τρόπο που αναπτύσσουμε λογισμικό.

ΠΗΓΕΣ

Άρθρα

[1]R. Geist, A. J. Outt, and F. Harris, \Estimation and enhancement of real-time software reliability through mutation analysis," IEEE Transactions on Computers

[13] Mike Papadakis and Nicos Malevris, An Empirical Evaluation of the First and Second Order Mutation Testing Strategies, Department of Informatics, Athens University of Economics and Business Athens, Greece

[2] A. T. Acree, On Mutation. PhD thesis, Georgia Institute of Technology, Atlanta GA

[3] W. E. Wong, On Mutation and Data Flow. PhD thesis, Purdue University, December 1993. (Also Technical Report SERC-TR-149-P, Software Engineering Research Center, Purdue University, West Lafayette, IN).

[4] R. Untch, A. J. Outt, and M. J. Harrold, Mutation analysis using program schemata," in Proceedings of the 1993 International Symposium on Software Testing, and Analysis, (Cambridge MA)

[5] A. J. Outt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, An experimental determination of sufficient mutation operators," ACM Transactions on Software Engineering Methodology

[6] W. E. Wong, M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, Constrained mutation in C programs," in Pro- ceedings of the 8th Brazilian Symposium on Software Engi- neering, (Curitiba, Brazil).

[7] A. J. Outt, G. Rothermel, and C. Zapf, An experimental evaluation of selective mutation," in Proceedings of the Fifteenth International Conference on Software Engineering, (Baltimore, MD) IEEE Computer Society Press.

[8]A. J. Outt and S. D. Lee, How strong is weak mutation?," in Proceedings of the Fourth Symposium on Soft- ware Testing, Analysis, and Verication, (Victoria, British Columbia, Canada).

[9] R. H. Untch, M. J. Harrold, and J. Outt, Schema-based mutation analysis." In preparation.

[10] A. J. Outt, R. Pargas, S. V. Fichter, and P. Khambekar, Mutation testing of software using a mimd computer,"

[11]V. N. Fleyshgakker and S. N. Weiss, “Efficient Mutation Analysis: A New Approach," in Proceedings of the International Symposium on Software Testing.

[12] A. P. Mathur and E. W. Krauser, \Mutant unication for improved vectorization," technical report SERC-TR-14-P, Software Engineering Research Center, Purdue University, West Lafayette IN.

Άλλες πήγες

[Wiki] Wikipedia link:
https://en.wikipedia.org/wiki/Mutation_testing

[V1] From Junit to mutation testing :
https://www.youtube.com/watch?v=9yG1c9Crnbk

[W1] Saarland University – Gordon Fraser – Mutation testing presentation
https://scholar.google.com/citations?user=PLpOpawAAAAJ&hl=en

[W2] Mutation testing in Software Testing: Mutant Score & Analysis Example
https://www.guru99.com/mutation-testing.html

[W3] Mutation testing Benefits & Types
https://www.tutorialspoint.com/software_testing_dictionary/mutation_testing.htm

[W4] Test Coverage is Dead – Long Live Mutation Testing
https://medium.com/appsflyer/tests-coverage-is-dead-long-live-mutation-testing-7fd61020330e

[W5] The University of Texas at Dallas – Mutation testing lecture
https://personal.utdallas.edu/~ewong/SE6367/03-Lecture/28-Mutation-Testing-two-slides-to-Page61.pdf

[W6] Muta-Pro: towards the definition of a mutation testing process
https://www.scielo.br/scielo.php?script=sci_arttext&pid=S0104-65002006000300005

[W7] Is Mutation Testing Ready to Be Adopted Industry-Wide?
https://www.researchgate.net/publication/309709540_Is_Mutation_Testing_Ready_to_Be_Adopted_Industry-Wide

[W8] Teaching software testing Concepts Using a mutation testing game
https://www.researchgate.net/publication/318127124_Teaching_Software_Testing_Concepts_Using_a_Mutation_Testing_Game

[W9] Demillo -Coupling effect
https://books.google.gr/books?id=hyaQobu44xUC&pg=PA594&lpg=PA594&dq=coupling+effect+demillo&source=bl&ots=R5lnYy9Oti&sig=ACfU3U2GfrPrKF79DdJaLA9DaKCiw_zz_g&hl=el&sa=X&ved=2ahUKEwiQxsy5wL3pAhXkxoUKHVEkDiAQ6AEwAXoECAsQAQ#v=onepage&q=coupling%20effect%20demillo&f=false

[W10] Investigations of the Software Testing Coupling Effect
https://cs.gmu.edu/~offutt/rsrch/papers/coupl.pdf