Ατομική Διπλωματική Εργασία

Πλαίσιο εργασίας για τη δημιουργία μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα για την γλώσσα C#.

Αντώνης Κουρράς

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Αύγουστος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πλαίσιο εργασίας για τη δημιουργία μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα για την γλώσσα C#.

Αντώνης Κουρράς

Επιβλέπων Καθηγητής
Ανδρέας Ανδρέου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς εκπλήρωση των απαιτήσεων απόκτησης Τίτλου Σπουδών Master σε Προηγμένες Τεχνολογίες Πληροφορικής στο Πανεπιστήμιο Κύπρου

Αύγουστος 2010

ΣΕΛΙΔΑ ΕΓΚΡΙΣΗΣ

Διατριβή Master

Τίτλος

Παρουσιάστηκε από

Αντώνης Κουρράς

Ερευνητικός Σύμβουλος

Ανδρέας Ανδρέου

Μέλος Επιτροπής

Γιώργος Χρυσάνθους

Μέλος Επιτροπής

Γιώργος Πάλλης

Πανεπιστήμιο Κύπρου

Αύγουστος, 2010

Ευχαριστίες

Νιώθω χρέος μου να ευχαριστήσω όλους τους ανθρώπους που με διάφορους τρόπους με βοήθησαν στην εκπόνηση αυτής της εργασίας.

Ο επιβλέπων καθηγητής Δρ. Ανδρέας Ανδρέου μου έδωσε την δυνατότητα να ασχοληθώ με το συγκεκριμένο θέμα. Οι κατευθυντήριες οδηγίες που μου έχει δώσει ήταν καθοριστικές στην επιτυχή ολοκλήρωση της εργασίας αυτής. Επίσης τον ευχαριστώ για την κατανόηση και την εμπιστοσύνη που μου επέδειξε καθ' όλη την διάρκεια της υλοποίησης της διπλωματικής μου εργασίας.

Ο Δρ. Αναστάσης Σοφοκλέους, αφιέρωσε πολύτιμο από τον χρόνο του έτσι ώστε να βοηθήσει καθ' όλη την διάρκεια της εκπόνησης της εργασίας. Οι γνώσεις του καθώς και ο τρόπος σκέψης του με βοήθησαν αρκετά έτσι ώστε να είμαι σε θέση να ολοκληρώσω με επιτυχία τη διπλωματική μου εργασία.

Τέλος, επειδή με αυτή την εργασία, ολοκληρώνονται και οι σπουδές μου ως μεταπτυχιακός φοιτητής, θα ήθελα να ευχαριστήσω την οικογένεια μου, που με υποστήριξε σε όλες μου τις αποφάσεις με κάθε τρόπο.

Περίληψη

Ο έλεγχος είναι μια διαδικασία η οποία είναι αναπόφευκτη για την ανάπτυξη και παραγωγή ενός συστήματος λογισμικού. Έχουν γίνει αρκετές προσεγγίσεις οι οποίες αναφέρονται στον έλεγχο συστημάτων λογισμικού. Με την αύξηση της πολυπλοκότητας του λογισμικού εμφανίστηκε η ανάγκη για εξεύρεση αξιόπιστων και αποδοτικών μεθόδων ελέγχου, έτσι ώστε να είναι επαρκής ο έλεγχος. Δηλαδή να μπορεί ο δημιουργός να εγγυηθεί την ορθότητα του εκάστοτε λογισμικού συστήματος.

Σε αυτή τη διπλωματική εργασία, έχει γίνει μια εκτενής μελέτη της πλατφόρμας Visual Studio 2010 καθώς και των δυνατοτήτων της ως προς την διαδικασία ελέγχου συστημάτων λογισμικού. Έχουν υλοποιηθεί δύο επαναχρησιμοποιήσιμα συστατικά, αυτό της στατικής ανάλυσης πηγαίου κώδικα για τη γλώσσα προγραμματισμού C# καθώς και της επαλήθευσης της ορθότητας του λογισμικού που βρίσκεται υπό έλεγχο. Επίσης, χρησιμοποιήθηκαν τα πιο πάνω συστατικά για την υλοποίηση μίας μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα γραμμένο στη γλώσσα προγραμματισμού C#. Οι μεταλλάξεις γίνονται σε επίπεδο μεθόδων. Επιπλέον, έχει προταθεί η βελτιστοποίηση της μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα έτσι ώστε να ενσωματωθεί «εξυπνάδα» στον τρόπο παραγωγής των μεταλλάξεων. Ακόμη, παρουσιάζονται διάφορα παραδείγματα για την αξιολόγηση και την παρουσίαση της χρήσης της πιο πάνω μηχανής στο χώρο της διαδικασίας ελέγχου συστημάτων λογισμικού.

Στην αρχή παρουσιάζεται το θεωρητικό υπόβαθρο καθώς και πληροφορίες για την έρευνα που έχει πραγματοποιηθεί αναφορικά με την πλατφόρμα Visual Studio 2010 και τη τεχνική ελέγχου βάσει μεταλλάξεων πηγαίου κώδικα. Στη συνέχεια η μεθοδολογία υλοποίησης και χρήσης των δύο συστατικά. Ακολούθως υπάρχει αναφορά στην αρχιτεκτονική και την ανάλυση της μηχανής παραγωγής μεταλλάξεων σε πηγαίο κώδικα.

Επίσης παρουσιάζονται διάφορα πειράματα για την επιβεβαίωση της εγκυρότητας και της χρησιμότητας της μηχανής παραγωγή μεταλλάξεων σε πηγαίο κώδικα. Τέλος παρουσιάζεται η αποτίμηση της εργασίας που έχει πραγματοποιηθεί καθώς και διάφορες προτάσεις για μελλοντική έρευνα στο χώρο.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Γενικά

Είναι ευρέως γνωστό πως στις μέρες μας όλο και περισσότερες διαδικασίες αυτοματοποιούνται. Η χρήση των Ηλεκτρονικών Υπολογιστών και η εκμετάλλευση των δυνατοτήτων τους βρίσκεται σε αρκετά ψηλά επίπεδα. Η ευρεία χρήση αυτή δημιουργεί την ανάγκη ύπαρξης ποιοτικών και αξιόπιστων συστημάτων λογισμικού. Ανάγκη η οποία γίνεται ακόμη πιο δύσκολη λόγω των πολύπλοκων και μεγάλης έκτασης λογισμικών συστημάτων που υπάρχουν σήμερα. Επιπλέον, η αγορά οδηγεί τις εταιρίες ανάπτυξης λογισμικού στην αύξηση της παραγωγικότητας σε περιορισμένη χρονική διάρκεια, με αντίχτυπο αρκετές φορές να μην επιτυγχάνεται η επιθυμητή ποιότητα.

Ένας σημαντικός παράγοντας της μειωμένης ύπαρξης ικανοποιητικής ποιότητας είναι η έλλειψη ή η μερική χρήση της διαδικασίας ελέγχου. Η διαδικασία ελέγχου ενός λογισμικού δεν θεωρείται καθόλου εύκολη λειτουργία. Αντιθέτως είναι μια επίπονη και δαπανηρή εργασία μιας και καταλαμβάνει το 25% - 50% του συνολικού κόστους στη διαδικασία παραγωγής ενός λογισμικού. Ακόμη ο χρόνος που χρειάζεται για τη διεκπεραίωσή της μπορεί να είναι και μεγαλύτερος από αυτόν του χρόνου κατασκευής του λογισμικού. Πρέπει να επισημανθεί πως ο έλεγχος αποσκοπεί στην εύρεση λαθών και όχι στην υπόδειξη έλλειψης λαθών.

Η διαδικασία ελέγχου αποτελείται από δύο φάσεις, τη φάση εντοπισμού λαθών (testing) και τη φάση αποσφαλμάτωσης (debugging). Η πιο χρονοβόρα φάση είναι η πρώτη όπου υπολογίζεται ότι χρειάζεται το 95% του χρόνου της διαδικασίας ελέγχου. Από αυτό συμπεραίνουμε ότι υπάρχει η ανάγκη καθώς και το ερευνητικό ενδιαφέρον δημιουργίας αλγορίθμων και κατ'επέκταση εργαλείων τα οποία θα αυτοματοποιούν την φάση αυτή και θα βοηθούν τους προγραμματιστές στο να εντοπίζουν και να ελέγχουν και να διορθώνουν τα σφάλματα στον πηγαίο κώδικά τους γρήγορα και αποτελεσματικά. Αυτό θα αυξήσει κατά πολύ την ποιότητα των παραγόμενων λογισμικών με αποτέλεσμα να κερδίζουν τόσο οι καταναλωτές όσο και οι εταιρίες παραγωγής λογισμικών συστημάτων.

1.2 Υποκίνηση Έρευνας

Μία πλατφόρμα η οποία τον τελευταίο καιρό έχει εξελιχθεί αρκετά με αποτέλεσμα να χρησιμοποιείται από πάρα πολλές εταιρίες ανάπτυξης λογισμικών συστημάτων είναι το Visual Studio. Το Visual Studio δίνει τη δυνατότητα στους χρήστες να υλοποιήσουν διαφόρων τύπων λογισμικά συστήματα όπως για παράδειγμα windows applications, web applications, services, classes κ.α. Ένα από τα θετικά του ότι είναι τόσο διαδεδομένη η συγκεκριμένη πλατφόρμα είναι η ύπαρξη αρκετών εξωτερικών υποβοηθητικών εργαλείων που κάνουν τη ζωή κάθε προγραμματιστή ευκολότερη. Επίσης αρκετά «bugs» τα οποία υπήρχαν στην αρχή της δημιουργίας της έχουν επιλυθεί με την συγκατάθεση όλων των χρηστών ανά τον κόσμο με αποτέλεσμα να γίνεται όλο και πιο αξιόπιστη και συνεργάσιμη με διάφορες άλλες πλατφόρμες και τεχνολογίες.

Τα πιο πάνω στοιχεία οδήγησαν την παρούσα διπλωματική στην επιμέρους μελέτη της πλατφόρμας Visual Studio 2010 και των δυνατοτήτων της. Επιπλέον η μείωση της χρήσης της γλώσσας προγραμματισμού Java δημιουργεί την ανάγκη της μεταφοράς του ερευνητικού ενδιαφέροντος της χρήσης της πιο πάνω πλατφόρμας σε πιο σύγχρονες και διαδεδομένες τεχνολογίες.

Στόχος της διπλωματικής αυτής εργασίας είναι η υλοποίηση μιας μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα, η οποία αποτελεί το βασικό στοιχείο στη δημιουργία της τεχνικής ελέγχου με τη χρήση μεταλλάξεων σε πηγαίο κώδικα (mutation testing). Η μηχανή αυτή χειρίζεται συστήματα λογισμικού γραμμένα σε C#

και στην πλατφόρμα Visual Studio 2010. Για την υλοποίηση του πιο πάνω εργαλείου έχουν υλοποιηθεί δύο επαναχρησιμοποιήσιμα συστατικά, τα οποία έχουν τη δυνατότητα να αποτελέσουν την ραχοκοκαλιά επόμενων μελετών ελέγχου συστημάτων λογισμικών. Τα συστατικά αυτά δίδουν τη δυνατότητα στο χρήστη να επιβεβαιώσει την εγκυρότητα ενός συστήματος λογισμικού γραμμένο στη πλατφόρμα Visual Studio, καθώς και την εκτέλεση στατικής ανάλυσης πηγαίου κώδικα γραμμένο σε γλώσσα προγραμματισμού C# για την εξαγωγή χρήσιμων πληροφοριών με τις οποίες θα μπορεί να επεξεργαστεί τον πηγαίο κώδικα.

Κεφάλαιο 2

Θεωρητικό Υπόβαθρο

2.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα μελετήσουμε τη θεωρία πίσω από την πράξη. Όπως και σε άλλους τομείς έτσι και στην Τεχνολογία Λογισμικού πρέπει να γίνει μια αρχική μελέτη των θεωριών έτσι ώστε να υπάρξει ένα γερό υπόβαθρο της συγκεκριμένης γνώσης. Με την γνώση αυτή θα είναι το πρώτο βήμα για την δημιουργία ενός αλγορίθμου υλοποίησης μηχανής μεταλλάξεων εντολών σε πηγαίο κώδικα ο οποίος θα επιφέρει καλά αποτελέσματα.

Στόχος αυτού του κεφαλαίου είναι η παρουσίαση σημαντικών εννοιών τόσο ως προς τον έλεγχο λογισμικού, την μετάλλαξη πηγαίου κώδικα (code mutation), τον έλεγχο συστήματος λογισμικού με τη χρήση μεταλλάξεων εντολών σε πηγαίο κώδικα (mutation testing), τους διάφορους τρόπους αναπαράστασης κώδικα καθώς και τις γλώσσες περιγραφής προδιαγραφών ενός λογισμικού συστήματος.

2.2 Μέθοδοι Ελέγχου

Βάσει της έρευνας [1] η οποία έχει γίνει με θέμα τις μεθόδους ελέγχου κατέληξε στην ύπαρξη τριών τεχνικών. Οι τεχνικές αυτές είναι:

Έλεγχος Μαύρου κουτιού (Black Box Testing): Ο Black-box έλεγχος ή concrete box είναι μια κατηγορία, όπου οι συνθήκες ελέγχου για το πρόγραμμα, δημιουργούνται με βάσει τις λειτουργίες και τις προδιαγραφές του συστήματος. Ο προγραμματιστής το

μόνο που χρειάζεται είναι πληροφορίες για τις τιμές εισόδου. Στη συνέχεια γνωρίζοντας αφαιρετικά την λειτουργία του συστήματος, αγνοώντας τον κώδικα (για αυτό λέγεται μαύρο κουτί), παρατηρεί τα αποτελέσματα και τα συγκρίνει με τα επιθυμητά. Η τεχνική αυτή βασίζεται στις προδιαγραφές και τις λειτουργίες του προγράμματος. Λόγω του ότι δεν χρειάζεται γνώση στον κώδικα του λογισμικού, δεν είναι αναγκαίο ο ελεγκτής να είναι ο ίδιος ο προγραμματιστής. Πιο κάτω γίνεται αναφορά στα κύρια πλεονεκτήματα καθώς και μειονεκτήματα της τεχνικής αυτής.

Πλεονεκτήματα:

- Είναι αποτελεσματικότερος σε μεγάλα κομμάτια κώδικα σε σχέση με την τεχνική ελέγχου άσπρου κουτιού (white box testing).
- Ο ελεγκτής δεν χρειάζεται προγραμματιστικές γνώσεις, αρκεί να γνωρίζει για συγκεκριμένες εισόδους τις επιθυμητές εξόδους.
- Ο προγραμματιστής και ο ελεγκτής είναι ανεξάρτητοι ο ένας απ’ τον άλλο.
- Βοηθά στο να ανακαλυφθούν οποιεσδήποτε ασάφειες ή ασυνέπειες στις προδιαγραφές.

Μειονεκτήματα:

- Μόνο ένας μικρός αριθμός πιθανών εισόδων μπορεί πραγματικά να ελεγχθεί μιας και είναι αδύνατο να ελεγχθούν όλοι οι δυνατοί συνδυασμοί. Αποτέλεσμα αυτού είναι ο μη ικανοποιητικός έλεγχος μιας και δεν ελέγχονται όλα τα πιθανά μονοπάτια.
- Για να δημιουργηθούν οι περιπτώσεις δοκιμής πρέπει να έχουμε σαφείς προδιαγραφές (μη διφορούμενες, συγκεκριμένες).
- Μπορεί να υπάρξει περιττή επανάληψη των εισόδων ελέγχου (test inputs), αν ο ελεγκτής δεν ενημερώνεται για τις περιπτώσεις δοκιμής (test cases) του προγραμματιστή.
- Δεν μπορεί να κατευθυνθεί προς συγκεκριμένα τμήματα του κώδικα που μπορούν να είναι πολύ σύνθετα και επομένως επιτρέπονται περισσότερα λάθη.
- Οι περισσότερες έρευνες που αφορούν τον έλεγχο έχουν κατευθυνθεί προς την τεχνική ελέγχου άσπρου κουτιού (white box testing).

Έλεγχος Άσπρου κουτιού (White Box Testing): Η τεχνική αυτή προϋποθέτει πως ο ελεγκτής πρέπει να γνωρίζει τις εσωτερικές λειτουργίες του προγράμματος, δηλαδή να γνωρίζει τη λειτουργία του καθώς και τον πηγαίο κώδικα. Οι δοκιμές ελέγχου σχεδιάζονται με βάσει τη δομή του προγράμματος. Λόγω του ότι τα δεδομένα ελέγχου όπως αναφέρθηκε και πιο πάνω λαμβάνονται βάσει του πηγαίου κώδικα, υπάρχει μεγαλύτερη πιθανότητα εντοπισμού λαθών.

Πλεονεκτήματα:

- Αυτή η τεχνική αναγνωρίζει με σχετική ευκολία το φαινόμενο της συμπτωματικής ακρίβειας (coincidental correctness). Το φαινόμενο αυτό αναφέρεται στις περιπτώσεις όπου παρόλο που το αποτέλεσμα προέκυψε ορθό, ο τρόπος που υπολογίζεται είναι λανθασμένος. Για τα συγκεκριμένα δεδομένα ελέγχου δεν είναι και τόσο σημαντικό αλλά κατά πάσα πιθανότητα για άλλα να μην είναι ορθό το αποτέλεσμα.
- Επίσης με την τεχνική αυτή μπορούμε να ελέγξουμε όλους τους πιθανούς τρόπους λειτουργίας του μιας και ο έλεγχος γίνεται βάσει του πηγαίου κώδικα.
- Με την τεχνική αυτή μπορούμε να ανακαλύψουμε περιπτώσεις νεκρού κώδικα, δηλαδή περιπτώσεις κώδικα που δεν εκτελούνται ποτέ και δεν μπορούν να ανακαλυφθούν με την τεχνική ελέγχου μαύρου κουτιού (black box testing).

Μειονεκτήματα:

- Με την τεχνική αυτή είναι πολύ δύσκολο να εντοπίσουμε λάθη λογικής.
- Είναι σχεδόν αδύνατο σε περιπτώσεις μεγάλων συστημάτων να εξεταστούν όλα τα μονοπάτια του κώδικα έτσι ώστε να ανακαλυφθούν όλα τα λάθη που υπάρχουν. Αυτό είναι ένα πρόβλημα το οποίο περιορίζει αρκετά την τεχνική αυτή και γίνονται προσπάθειες επίλυσής του.
- Δεν υπάρχει τρόπος να εντοπισθούν τα μονοπάτια τα οποία έχουν παραλειφθεί.

Έλεγχος Γκρίζου Κουτιού (Gray Box Testing): Η τεχνική αυτή είναι ένας συνδυασμός των δύο πιο πάνω τεχνικών. Δηλαδή συνδυάζει τόσο τον έλεγχο βάσει της εσωτερικής δομής όσο και τον έλεγχο βάσει των προδιαγραφών του λογισμικού. Σκοπός της δημιουργίας της πιο πάνω τεχνικής είναι η όσο το δυνατόν καλύτερη αξιοποίηση των

πλεονεκτημάτων των δυο προηγούμενων τεχνικών, με αποτέλεσμα τον καλύτερο έλεγχο.

2.3 Μετάλλαξη Πηγαίου Κώδικα (Code Mutation)

Η τεχνική μετάλλαξης [7] αποσκοπεί στην δημιουργία μεταλλαγμένων προγραμμάτων βασισμένα σε ένα αρχικό πηγαίο κώδικα. Κάθε αλλαγή είναι ατομική έτσι ώστε το μεταλλαγμένο πρόγραμμα να μην απέχει κατά πολύ από το αρχικό. Όλες οι αλλαγές διέπονται από την γραμματική της εκάστοτε γλώσσας προγραμματισμού. Καθορίζονται από τους τελεστές μετάλλαξης (mutation operators) οι οποίοι διαφέρουν σχετικά με την γλώσσα προγραμματισμού που είναι γραμμένος ο πηγαίος κώδικας.

Υπάρχουν διάφοροι τύποι τελεστών μετάλλαξης. Για παράδειγμα σε μία αντικειμενοστρεφή γλώσσα προγραμματισμού υπάρχουν:

- Οι παραδοσιακοί τελεστές (traditional) [5], όπου αναφέρονται σε αλλαγές επιπέδου μεθόδων (method-level based). Δηλαδή σε αλλαγές οι οποίες μπορούν να εφαρμοστούν εντός μίας μεθόδου. Μερικές αλλαγές αναφέρονται σε αριθμητικές, λογικές, σχεσιακές και άλλες πράξεις.
- Οι τελεστές επιπέδου κλάσης (class-level) [4], όπου αναφέρονται σε αλλαγές οι οποίες μπορούν να εφαρμοστούν σε μία κλάση ή μεταξύ δύο ή περισσότερων κλάσεων. Μερικές αλλαγές αναφέρονται σε αντικατάσταση κλήσης μιας μεθόδου με μια άλλη παρόμοια, είτε αλλαγή των δικαιωμάτων πρόσβασης (public, static, private etc).

Παραδείγματα εφαρμογής μεταλλάξεων πηγαίου κώδικα:

Μετάλλαξη σε επίπεδο μεθόδου:

Αρχικό πρόγραμμα
1 int max(int x, int y) **2 {** **3 int mx = x;** **4 if (x > y)** **5 mx = x;** **6 else** **7 mx = y;** **8 return mx;** **9 }**

Μεταλλαγμένο πρόγραμμα
1 int max(int x, int y) **2 {** **3 int mx = x;** **4 if (x < y)** **5 mx = x;** **6 else** **7 mx = y;** **8 return mx;** **9 }**

Μετάλλαξη σε επίπεδο κλάσης:

Αρχικό πρόγραμμα
1 public void foo() **2 {** **3 ….** **4 }**

Μεταλλαγμένο πρόγραμμα
1 private void foo() **2 {** **3 ….** **4 }**

Υπάρχουν αρκετοί τρόποι όπου μπορεί να χρησιμοποιηθεί η τεχνική αυτή. Για παράδειγμα χρησιμοποιείται για τη δημιουργία πολλαπλών προγραμμάτων με σκοπό την περαιτέρω ανάλυσή τους έτσι ώστε να βρεθεί το λάθος στο αρχικό σύστημα. Επίσης χρησιμοποιείται για την δημιουργία προγραμμάτων τα οποία θα βοηθήσουν στον έλεγχο καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής. Περαιτέρω επεξήγηση της χρησιμοποίησης της πιο πάνω τεχνικής καθώς και παραδείγματα χρησιμοποίηση της θα υπάρξει στο Κεφάλαιο 4 καθώς και στην επεξήγηση των δυνατοτήτων της υλοποιημένης μηχανής παραγωγής μεταλλάξεων στο Κεφάλαιο 5.

Σε αυτή την διπλωματική εργασία θα ασχοληθούμε με τη δημιουργία μιας μηχανής παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα. Επίσης οι τελεστές μετάλλαξης που θα χρησιμοποιηθούν θα είναι αποκλειστικά σε επίπεδο μεθόδων, δηλαδή οι παραδοσιακοί τελεστές.

2.4 Έλεγχος πηγαίου κώδικα βάσει μεταλλάξεων (Mutation Testing)

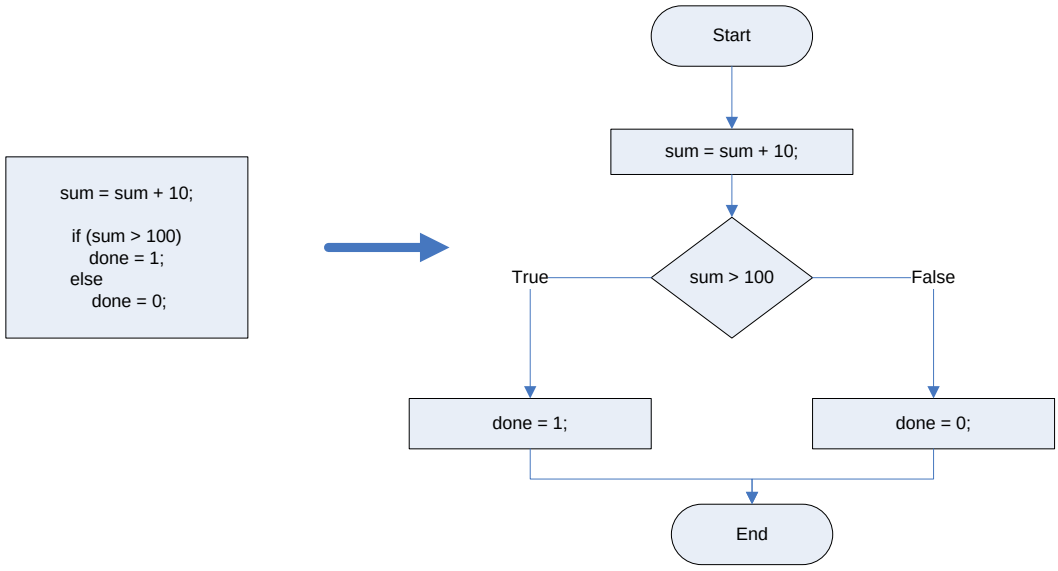
Το έλεγχος πηγαίου κώδικα βάσει μεταλλάξεων (mutation testing) [6] είναι μία τεχνική ελέγχου (white box) την οποία εισήγαγαν οι Hamlet [2] και DeMillo [3] και βασίζεται στην εισαγωγή λαθών σε ένα πρόγραμμα, βάσει της τεχνικής μετάλλαξης πηγαίου κώδικα (code mutation) που αναφέρθηκε πιο πάνω. Επιπλέον παρέχει ένα κριτήριο ελέγχου που ονομάζεται βαθμός καταλληλότητας μεταλλάξεων (mutation adequacy score). Το κριτήριο αυτό χρησιμοποιείται για την μέτρηση της αποτελεσματικότητας ενός συνόλου ελέγχου με βάσει την ικανότητα του να εντοπίζει λάθη. Η γενική ιδέα που διέπει τον έλεγχο μετάλλαξης είναι ότι τα λάθη που εισάγονται αντιπροσωπεύουν τα λάθη που γίνονται συχνά από τους προγραμματιστές. Επιπλέον βασίζεται και στη θεωρία πως οι προγραμματιστές τείνουν να δημιουργούν προγράμματα αρκετά κοντά στην ορθή έκδοση τους. Τέτοια λάθη εμφυτεύονται σκόπιμα στο αρχικό πρόγραμμα για τη δημιουργία ενός συνόλου από λανθασμένα προγράμματα τα ονομαζόμενα μεταλλαγμένα (mutants), κάθε ένα από τα οποία περιέχει μια ξεχωριστή αλλαγή. Για την αποτίμηση της ποιότητας ενός δοθέντος συνόλου ελέγχου τα μεταλλαγμένα προγράμματα εκτελούνται με αυτό το σύνολο για να ελεγχθεί κατά πόσο τα εμφυτευμένα λάθη μπορούν να εντοπιστούν. Στόχος είναι η αποτίμηση του βαθμού καταλληλότητας του συνόλου περιπτώσεων δοκιμής (test cases) που χρησιμοποιείται για τον έλεγχο του λογισμικού συστήματος.

2.5 Αναπαραστάσεις Κώδικα

Υπάρχουν αρκετοί τρόποι αναπαράστασης πηγαίου κώδικα. Ο λόγος που χρειαζόμαστε αυτούς τους τρόπους είναι για καλύτερη κατανόηση άλλα και διαχείριση του πηγαίου κώδικα. Μερικοί από αυτούς τους τρόπους είναι με τη χρήση γράφων και δυαδικών δέντρων. Οι πιο γνωστοί γράφοι είναι ο γράφος ροής ελέγχο υ [9] και ο γράφος εξαρτήσεων [10]. Με την χρήση των γράφων αυτών, οι προγραμματιστές μπορούν να δείξουν την λειτουργία και τον τρόπο που δουλεύει το λογισμικό και έτσι διευκολύνεται ο έλεγχος του λογισμικού, η συντήρηση του καθώς και η περαιτέρω ανάπτυξη του. Η δο μή των γράφων, σύνολο από κόμβους και ακμές καθώς και η κατεύθυνση που μπορούμε να τους δώσουμε βοηθούν στην αναπαράσταση του κώδικα. Πιο κάτω θα παρουσιαστούν οι πιο γνωστοί γράφοι αναπαράστασης κώδικα.

Ο γράφος ροής ελέγχου [9] είναι μία γραφική αναπαράσταση όλων των πιθανών μονοπατιών τα οποία μπορούν να προκύψουν από το αντίστοιχο πρόγραμμα κατά τη διάρκεια της εκτέλεσης τους. Αποτελείται από κόμβους, όπου κάθε κόμβος αντιπροσωπεύει μία εντολή του προγράμματος. Ο γράφος είναι κατευθυνόμενος και από τις κατευθύνσεις των ακμών μπορούμε να παρατηρήσουμε τις μεταπηδήσεις της ροής ελέγχου.

Ο γράφος αυτός δημιουργείται μέσω ιδικών προγραμμάτων (Walkers/Visitors) τα οποία διαπερνούν τον κώδικα, αναγνωρίζοντας τις διάφορες εντολές. Όπως προαναφέρθηκε, για κάθε εντολή δημιουργείται ένα κόμβος και μια ακμή η οποία τον ενώνει με τον προηγούμενο. Σε περίπτωση εντολής διακλάδωσης τότε οι ακμές που δημιουργούνται είναι δύο.
Στο Σχήμα 2.1 παρατηρούμε ένα παράδειγμα γράφου ροής δεδομένου βάσει του κώδικα που παρουσιάζει.



Σχήμα 2.1 –Γράφος Ροής Δεδομένων (CFG)

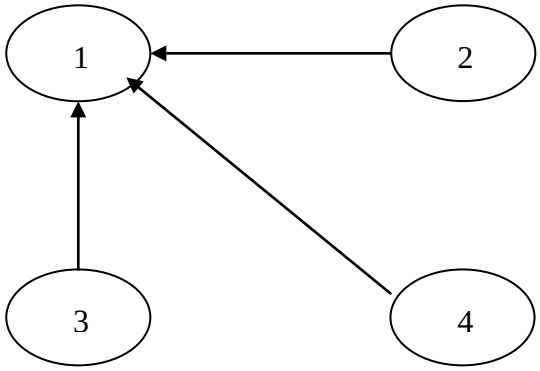
Ο γράφος εξαρτήσεων [10] είναι μία γραφική αναπαράσταση όλων των εξαρτήσεων που υπάρχουν σε ένα πρόγραμμα. Μια ακμή από τον κόμβο Vi στον κόμβο Vj υποδηλώνει πως ο υπολογισμός ή η πράξη που γίνεται στον κόμβο Vi εξαρτάται άμεσα από την τιμή που υπολογίστηκε στον κόμβο Vj. Για την ακρίβεια, δηλώνει πως ο υπολογισμός ή η πράξη που γίνεται στον κόμβο Vi χρησιμοποιεί μια μεταβλητή Var

που προσδιορίζεται η τιμή της στον κόμβο Vj και υπάρχει μονοπάτι από τον Vj στον Vi στο οποίο η τιμή της μεταβλητής Var δεν επαναπροσδιορίζεται.

Στο Σχήμα 2.2 παρατηρούμε ένα παράδειγμα γράφου ροής εξαρτήσεων βάσει του κώδικα που παρουσιάζει.

```
1       int x = 5;

2       if (x<6)
3       z = x + 3;
else
4       z = x - 3;
```



Σχήμα 2.2 –Γράφος Εξαρτήσεων (DDG)

2.6 Γλώσσες Περιγραφής Προδιαγραφών Συστήματος

Κάθε λογισμικό σύστημα, αποσκοπεί στο να εξυπηρετήσει μία ανάγκη με το να δημιουργήσει μια αυτοματοποιημένη διεργασία. Για να επιτυγχάνεται αυτό κάθε λογισμικό διέπεται από ένα σύνολο από προδιαγραφές, οι οποίες όταν καλυφθούν τότε το λογισμικό χαρακτηρίζεται ως αξιόπιστο μιας και εκτελεί την διεργασία για την οποία έχει δημιουργηθεί. Ένα από τα σημαντικά βήματα που έχουν γίνει, είναι η κωδικοποίηση των προδιαγραφών και η ενσωμάτωσή τους στον πηγαίο κώδικα με την χρήση γλωσσών περιγραφής προδιαγραφών. Η χρήση αυτών των γλωσσών είναι σημαντική για την παραγωγή σεναρίων ελέγχου, για τον πρόωρο εντοπισμό λογικών σφαλμάτων αλλά και για την παραγωγή αναμενόμενων αποτελεσμάτων από την εκτέλεση του υπό έλεγχο λογισμικού συστήματος. Ο προγραμματιστής έχει τη δυνατότητα να καταγράψει της προδιαγραφές ως ένα σύνολο από προαπαιτούμενα (pre-conditions), σταθερές (invariants) και αναμενόμενα (post-conditions). Μία τέτοια γλώσσα είναι και τα Code Contracts [13] που παρέχονται στην πλατφόρμα Visual Studio 2010 και θα αναλυθούν στην συνέχεια. Σε προηγούμενες εκδόσεις η γλώσσα αυτή ονομαζόταν Spec# [12]. Σε αυτή τη διπλωματική εργασία θα γίνει μια εκτενής

αναφορά στα Code Contracts, τις δυνατότητες που παρέχουν καθώς και πως μπορούν να συνδυαστούν με την τεχνική ελέγχου βάσει μεταλλάξεων (mutation engine).

Κεφάλαιο 3

Παρουσίαση Πλατφόρμας Visual Studio 2010 και των δυνατοτήτων που παρέχει ως προς τον έλεγχο λογισμικών συστημάτων.

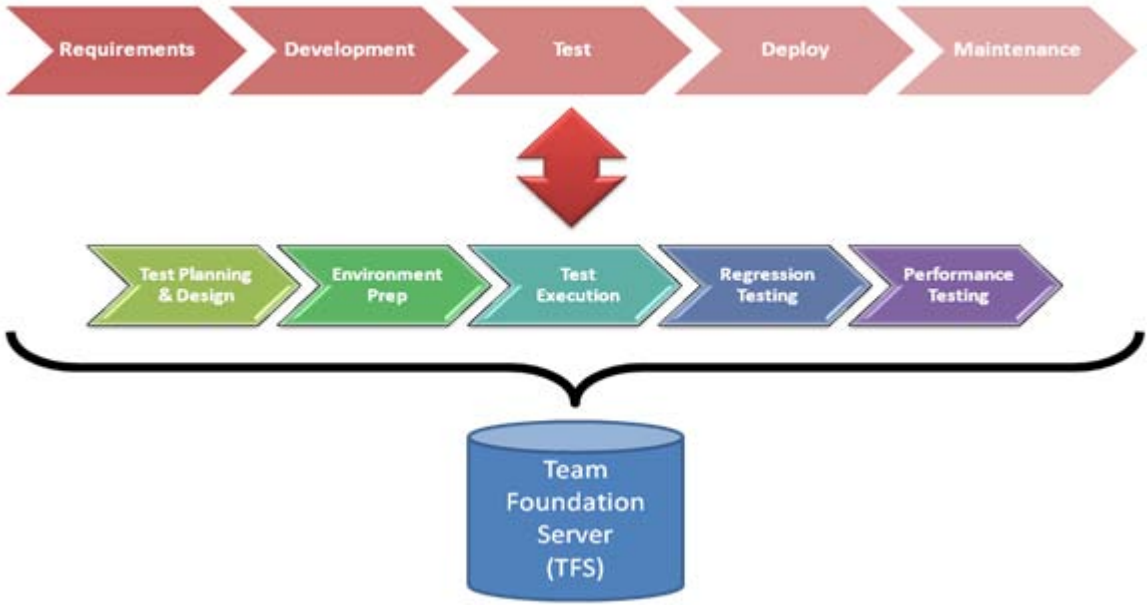
3.1 Εισαγωγή

Η πλατφόρμα Visual Studio 2010 αποσκοπεί στην παροχή αρκετών δυνατοτήτων οι οποίες έχουν ως απώτερο σκοπό την απλοποίηση της διαδικασίας ανάπτυξης ενός λογισμικού συστήματος από την φάση του σχεδιασμού μέχρι την υλοποίηση και τον έλεγχό του. Υποστηρίζει ένα μεγάλο φάσμα τεχνολογιών, συγκεντρωμένες σε μία πλατφόρμα, πράγμα που το κάνει πιο ελκυστικό. Μέσω της συγκεκριμένης πλατφόρμας μπορούν να δημιουργηθούν λογισμικά προγράμματα σε διάφορες γλώσσες προγραμματισμού, μερικές από αυτές είναι οι εξής:

- C #
- C ++
- Visual Basic
- ASP.NET

- SharePoint

Το πιο κάτω σχήμα παρουσιάζει το όραμα της πλατφόρμας η οποία στοχεύει στο να κάνει πιο εύκολη τη ζωή όλων των εμπλεκόμενων για την ανάπτυξη ενός λογισμικού συστήματος. Επίσης αποσκοπεί στην χρησιμοποίηση του τόσο από προγραμματιστές (developers) όσο και από ελεγκτές (testers) με δυνατότητα επικοινωνίας και εξαγωγής χρήσιμων συμπερασμάτων.



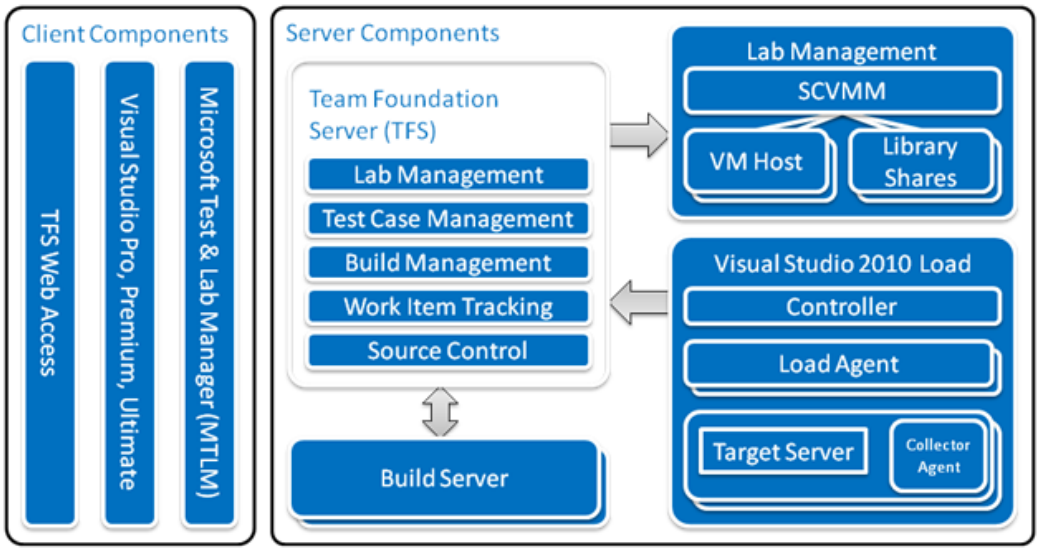
Σχήμα 3.1 – Πως το V.S οραματίζεται το περιβάλλον ελέγχου μέσω τις ίδιας της πλατφόρμας

Σε αυτή τη διπλωματική εργασία έγινε μια μελέτη ως προς τις δυνατότητες της πλατφόρμας Visual Studio 2010 αναφορικά με τον έλεγχο του υπό ανάπτυξη λογισμικού συστήματος.

3.2 Συστατικά Ελέγχου (Test Components)

Στο πιο κάτω σχήμα παρουσιάζονται τα κυριότερα μέρη του Πλαισίου Ελέγχου (Test Framework) [14] τα οποία συμβάλουν στην διαδικασία ελέγχου βάσει του ρόλου που έχουν. Υπάρχουν τρία εργαλεία (client tools) όπου οι χρήστες μπορούν να χρησιμοποιήσουν για να τρέξουν ελέγχους τόσο στο δικό τους περιβάλλον όσο και εξ

αποστάσεως, καθώς και να διαχειριστούν πληροφορίες από ελέγχους που ήδη έχουν γίνει. Τα εργαλεία αυτά επικοινωνούν απευθείας με το TFS το οποίο αποτελεί την καρδιά του συστήματος. Ο εξυπηρετητής «build server» επικοινωνεί με τα κομμάτια Lab manager και Test Case Management μέσω του TFS με σκοπό την εκτέλεση αυτοματοποιημένων ελέγχων ως μέρος της διαδικασίας «build process».



Σχήμα 3.2 – Συστατικά ελέγχου της πλατφόρμας Visual Studio 2010

3.3 Πλαίσιο Εργαλείων Ελέγχου (Test Tools Framework)

Η πλατφόρμα Visual Studio 2010 παρέχει ένα σύνολο από εργαλεία ελέγχου τα οποία βοηθούν στην αύξηση της παραγωγικότητας κατά την διάρκεια του κύκλου ζωής ελέγχου. Όλα αυτά τα εργαλεία είναι χτισμένα βάσει της αρχιτεκτονικής του Πλαισίου Εργαλείων Ελέγχου (Test Framework Tools) [14]. Το πλαίσιο (Framework) αυτό παρέχει τις δυνατότητες δημιουργίας, διαχείρισης, αλλαγής και εκτέλεσης ελέγχων καθώς και την εξαγωγή/αποθήκευση των σχετικών αποτελεσμάτων του ελέγχου που έχει πραγματοποιηθεί. Μία γκάμα από τύπους ελέγχων όπως Unit, Web, Load, Coded UI και Manual έλεγχοι σε συνδυασμό με μετρικές κάλυψης κώδικα συνυπάρχουν στην πλατφόρμα Visual Studio 2010.

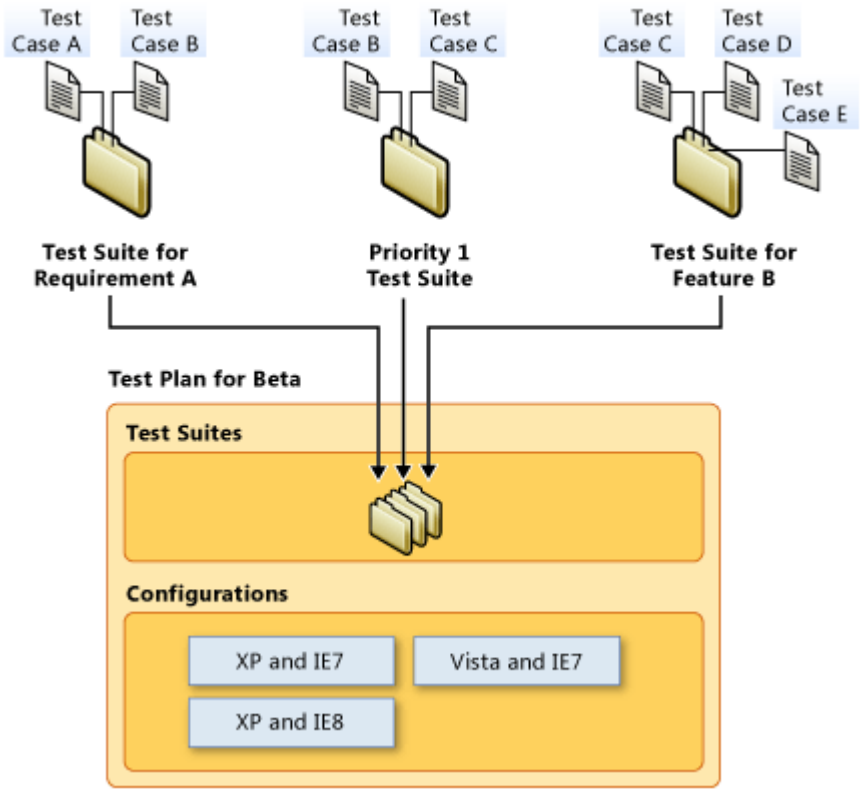
Ένα σημαντικό στοιχείο είναι η δυνατότητα του χρήστη να χρησιμοποιήσει τα εργαλεία αυτά, είτε ανεξάρτητα είτε συνδυασμένα. Επίσης η πλατφόρμα αποσκοπεί στο να εξαλείψει τα εμπόδια μεταξύ ελεγκτών (testers) και προγραμματιστών (developers) έτσι ώστε η συνεργασία τους να γίνεται αποδοτικότερη και αποτελεσματικότερη. Όλα τα

ενσωματωμένα εργαλεία ελέγχου έχουν το δικό τους API δίδοντας τη δυνατότητα στον χρήστη να τα χρησιμοποιήσει σε συνδυασμό ακόμη και με (3rd party) εξωτερικά εργαλεία ελέγχου.

Επίσης υπάρχει η δυνατότητα της δημιουργίας πλάνων ελέγχου (Test Plan) τα οποία θα κατευθύνουν την διαδικασία ελέγχου επιτρέποντας στον χρήστη να αποτιμήσει τον απαραίτητο χρόνο που θα χρειαστεί. Ένα πλάνο ελέγχου αποτελείται από:

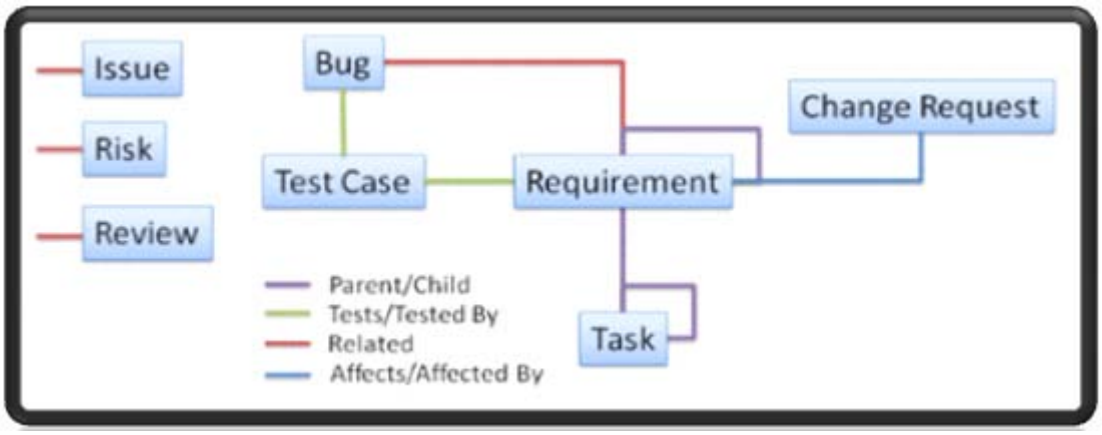
- Συλλογή ελέγχων (Test suites)
- Περιπτώσεις δοκιμής (Test Cases)
- Διαμόρφωση (Configuration)

Η χρήση κάθε επιμέρους στοιχείου διαφαίνεται στο πιο κάτω σχήμα, όπου μία συλλογή ελέγχων (test suite) αποτελείται από ένα σύνολο από περιπτώσεις δοκιμής (Test Cases). Ακόμη ένα πλάνο ελέγχου (Test Plan) αποτελείται από ένα σύνολο από συλλογές ελέγχων (Test Suites) και των αντίστοιχων ρυθμίσεων που χρειάζονται.



Σχήμα 3.3 – Πλάνο Ελέγχου (Test Plan)

Οι περιπτώσεις δοκιμής (Test Cases) αποθηκεύονται υπό μορφή “Work Items” στην αποθήκη TFS Work Item Store. Το Visual Studio 2010 μπορεί να χειριστεί αρκετά “Work Items”. Κάθε “Work Item” όπως παρουσιάζεται στο πιο κάτω σχήμα, μπορεί να συνδεθεί με άλλους τύπους συνδέσμων όπως “Parent/Child,” “Tests/TestedBy,” “Related,” “Affects/AffectedBy.”



Σχήμα 3.4 – Work Item formation

Ο χρήστης μπορεί να καθορίσει μέσω των ρυθμίσεων ελέγχου ποιούς τύπους δεδομένων θα συλλέγει καθώς και πώς θα επιδρά το σύστημα κατά την εκτέλεση των ελέγχων. Επίσης κατά την εμφάνιση πιθανών λαθών μπορεί ο χρήστης να υποβάλει πλήρη αναφορά για το περιστατικό χρησιμοποιώντας είτε το σύστημα «Test Runner» είτε από το ίδιο το Visual Studio.

3.4 Τύποι Εργαλείων Ελέγχου (Test Tools Types)

Η πλατφόρμα Visual Studio 2010 παρέχει κάποιους τύπους εργαλείων ελέγχου οι οποίοι μπορούν να χρησιμοποιηθούν για συγκεκριμένους σκοπούς ελέγχων. Επίσης δίνει τη δυνατότητα δημιουργίας μη προκαθορισμένων τύπων ελέγχου (custom test types) για την επίτευξη κάλυψης επιπλέον αναγκών ελέγχου. Μερικοί από τους προκαθορισμένους τύπους εργαλείων ελέγχου είναι οι εξής:

- Unit Tests:

Δίδεται η δυνατότητα στους προγραμματιστές και στους ελεγκτές να ελέγξουν γρήγορα και αποτελεσματικά την ύπαρξη λογικών σφαλμάτων στις μεθόδους των κλάσεων. Είναι αρκετά χρήσιμος τύπος ειδικά κατά τη διάρκεια δημιουργίας του κώδικα ή κατά την διάρκεια εφαρμογής αλλαγών σε ήδη υπάρχον σύστημα. Επίσης μπορεί να συνδυαστεί με τους τύπους Load και Performance Tests παρέχοντας δυνατότητα ελέγχου της αξιοπιστίας και αποδοτικότητας μιας εφαρμογής .

- Web Tests:

Αποτελείται από ένα σύνολο από HTTP αιτήσεις (requests) με σκοπό τον έλεγχο εφαρμογών διαδικτύου. Αυτός ο τύπος μπορεί να συνδυαστεί με Performance και Stress Tests.

- Load Tests

Κυρίαρχος λόγος ύπαρξης του τύπου αυτού είναι ο έλεγχος της συμφόρησης όταν αρκετοί χρήστες έχουν πρόσβαση σε ένα server την ίδια χρονική στιγμή. Ο τύπος αυτός μπορεί να συνδυαστεί με Web Tests καθώς και Load Tests.

- Performance Tests

Κυρίαρχος λόγος ύπαρξης του τύπου αυτού είναι ο έλεγχος της αποδοτικότητας του λογισμικού συστήματος. Ο τύπος αυτός αποτελείται από διάφορα Load Tests.

- Coded UI Tests

Αυτός ο τύπος ελέγχου αποσκοπεί στον αυτόματο έλεγχο γραφικών διαπροσωπιών. Ελέγχει την λειτουργικότητα των controls που αποτελούν το UI. Επιτυγχάνεται πιο γρήγορα ο έλεγχος μέσω αυτού του τύπου παρά με τον έλεγχο βάσει manual testing.

- Manual Tests

Ο τύπος αυτός χειρίζεται τα “Work Items” των Test Cases που αναφέρθηκε προηγουμένως.

- Generic Tests

Ο τύπος αυτός χρησιμοποιείται για τον έλεγχο εξωτερικών λογισμικών τα οποία δεν έχουν δημιουργηθεί από οποιουδήποτε πλατφόρμα Visual Studio.

- Ordered Tests

Ο τύπος αυτός αποτελείται από άλλους τύπους ελέγχων οι οποίοι πρέπει να τρέξουν με μία προκαθορισμένη σειρά έτσι ώστε να έχουν αξία. Τα αποτελέσματα μπορούν να εξαχθούν συνολικά αλλά και ανεξάρτητα για κάθε τύπο ελέγχου.

3.5 Ανάλυση Επιρροής Ελέγχου (Test Impact Analysis)

Η ανάλυση επιρροής ελέγχου (Test Impact Analysis) είναι μια δυνατότητα την οποία παρέχει η πλατφόρμα Visual Studio 2010 βάσει της οποίας ο χρήστης μπορεί να γνωρίζει ποιοί έλεγχοι επηρεάζονται από τις τελευταίες αλλαγές στον πηγαίο κώδικα οι οποίες δεν έχουν ελεγχθεί ακόμη. Για να μπορεί όμως να ισχύσει αυτή η δυνατότητα πρέπει το λογισμικό να έχει ήδη δημοσιευμένα αποτελέσματα ελέγχου στο «Team Build» από την προηγούμενη ολοκληρωμένη με επιτυχία εκτέλεση. Έτσι με τη βοήθεια αυτού του εργαλείου αποφεύγονται οι αχρείαστες εκτελέσεις αρκετών ελέγχων που περιλαμβάνει το test suite και εστιάζεται η προσοχή του χρήστη μόνο στους ελέγχους οι οποίοι έχουν επηρεαστεί από τις αλλαγές στον κώδικα. Ειδικά όταν το σύνολο των ελέγχων είναι αρκετά μεγάλο, ο χρόνος που εξοικονομείται είναι αρκετός και μπορεί να αξιοποιηθεί διαφορετικά.

3.6 Κάλυψη Πηγαίου Κώδικα (Code Coverage)

Με την κάλυψη πηγαίου κώδικα (Code Coverage) [15] παρέχεται η δυνατότητα στον χρήστη να κατανοήσει ποιο μέρος του κώδικα πραγματικά έχει τρέξει και ποιο όχι. Συνδυάζεται με την δημιουργία και την εκτέλεση διαφόρων περιπτώσεων δοκιμής (test cases) τα οποία καθορίζει ο χρήστης. Μία λεπτομερής αναφορά παράγεται στο τέλος

της κάλυψης πηγαίου κώδικα παρέχοντας στον χρήστη τις απαραίτητες πληροφορίες. Επίσης το η κάλυψη πηγαίου κώδικα (Code Coverage) [15] στο Visual Studio 2010 πλέον δεν αναφέρεται μόνο σε μία εκτέλεση ελέγχου αλλά αντιθέτως μπορεί να συνδυάσει τα αποτελέσματα από κάθε έλεγχο, παρουσιάζοντας τα στον χρήστη συνολικά. Τα αποτελέσματα περιέχουν γραμμές ή «blocks» τα οποία έχουν διαπεραστεί κατά την εκτέλεση συγκεκριμένων ελέγχων.

3.7 Συμβόλαια Πηγαίου Κώδικα (Code Contracts)

Τα Συμβόλαια Πηγαίου Κώδικα (Code Contracts) [13] χρησιμοποιούνται για να κωδικοποιούμε τις προδιαγραφές σε ένα λογισμικό το οποίο αναπτύσσεται στην πλατφόρμα Visual Studio 2010. Αποτελούνται από προαπαιτούμενα (pre-conditions), αναμενόμενα (post-conditions) και σταθερές (invariants). Συμπεριφέρονται ως ένα ελεγχόμενο αρχείο των εσωτερικών και εξωτερικών APIs του συστήματος. Στόχος της ύπαρξης τους είναι η βελτίωση της διαδικασίας ελέγχου μέσω της χρήση κατά την εκτέλεση ελέγχου (runtime checking), στατικής επιβεβαίωσης μη παραβίασης των συμβόλων πηγαίου κώδικα (static contract verification) και δημιουργίας εγγράφων (documentation) generation. Ουσιαστικά με τη χρήση των Συμβολαίων Πηγαίου Κώδικα (Code Contracts) [13] συνεπάγεται η επίτευξη όλων των προδιαγραφών που έχουν τεθεί πριν ξεκινήσει να αναπτύσσεται το λογισμικό.

Όπως αναφέρθηκε και πιο πάνω, υπάρχουν 3 τρόποι βάσει των οποίων συμβάλει η παρουσίαση των Συμβολαίων Πηγαίου Κώδικα (Code Contracts) στην βελτίωση του ελέγχου ενός λογισμικού:

- Κατά την εκτέλεση έλεγχος (Runtime Checking):

  Ο έλεγχος καταπάτησης ή όχι των προαπαιτούμενων (pre-conditions), αναμενόμενων (post-conditions) και σταθερών (invariants) πραγματοποιείται κατά την εκτέλεση του προγράμματος. Σε περίπτωση που παραβιάζεται μια συνθήκη του συμβολαίου πηγαίου κώδικα (code contract) συμπεριφέρεται ως μια διαμαρτυρία (assertion) με μήνυμα εκτός και αν επιλέξει ο χρήστης να το «πνίξει».

- Στατικός Έλεγχος (Static Checking):

Ο έλεγχος παραβίασης ή όχι των προαπαιτούμενων (pre-conditions), αναμενόμενων (post-conditions) και σταθερών (invariants) πραγματοποιείται στατικά κατά την διάρκεια ανάπτυξης του προγράμματος. Φυσικά οι έλεγχοι είναι περιορισμένοι μιας και αρκετοί έλεγχοι μόνο κατά την ώρα της εκτέλεσης μπορούν να γίνουν. Είναι όμως αρκετά βοηθητικό για να αντιμετωπίσουμε πιθανά σφάλματα πριν καν τρέξουμε το πρόγραμμα. Η συμπεριφορά αυτού του είδους ελέγχου είναι διαφορετική από την προηγούμενη. Σε αυτού του τύπου ελέγχους οι καταπατήσεις των προδιαγραφών εμφανίζονται υπό την μορφή προειδοποιήσεων (warnings), επίσης προτείνονται αλλαγές στα συμβόλαια πηγαίου κώδικα (code contracts) σε περίπτωση που ο κώδικας δεν συμφωνεί με αυτά οδηγώντας τον χρήστη στον έλεγχό τους για επικύρωση της εγκυρότητας τους.

- Documentation Generation:

Παράγεται ένα XML αρχείο με πληροφορίες από τα Code Contracts.

3.8 Ανάλυση Πηγαίου Κώδικα (Code Analysis)

Η πλατφόρμα Visual Studio 2010 παρέχει τη δυνατότητα ανάλυσης του κώδικα καθώς και δημιουργίας κανόνων ανάλυσης κώδικα από τον ίδιο το χρήστη βάσει των αναγκών του. Οι ρόλοι μπορεί να αναφέρονται σε διάφορους τομείς, όπως σχεδιασμό, σημασιολογία κ.α. Το εργαλείο αυτό ελέγχει τους κανόνες αυτούς και παρουσιάζει σε μορφή προειδοποιήσεων, πιθανών παραβιάσεών τους. Η ανάλυση μπορεί να γίνεται με κάθε μεταγλώττιση του συστήματος αυτόματα ή χειροκίνητα από τον χρήστη όποτε το επιθυμεί. Ο χρήστης έχει τη δυνατότητα να επιλέξει ένα σύνολο από προκαθορισμένους ή και δικούς του κανόνες.

3.9 Γράφοι και Διαγράμματα (Graphs and Diagrams)

Αρκετές φορές για την καλύτερη/ευκολότερη κατανόηση ενός άγνωστου κώδικα η χρήση γραφικών αναπαραστάσεων είναι αρκετά βοηθητική. Κρίνεται βοηθητική γιατί περιγράφει την δομή του εν λόγω λογισμικού, πράγμα δύσκολο να κατανοήσει κάποιος σε ένα μεγάλο και πολύπλοκο λογισμικό. Έτσι και η πλατφόρμα Visual Studio 2010 παρέχει τη δυνατότητα παραγωγής γραφημάτων (γράφους και διαγράμματα) τα οποία συσχετίζονται με τον τρέχοντα κώδικα ο οποίος είναι υπό ανάπτυξη ή έλεγχο.

Μερικά από τα γραφήματα για τα οποία δίδεται η δυνατότητα δημιουργίας τους μέσω της πλατφόρμας είναι:

- Γράφος Εξάρτησης (Dependence Graph)

Γράφος ο οποίος αναπαριστά συσχετιζόμενα δεδομένα (αλληλοεξαρτώμενα) μετά από ανάλυση του κώδικα. Η σχετική ανάλυση μπορεί να γίνει σε επίπεδο «assembly», «namespace», «class», «types», «methods», «externals» ακόμη και σε συνδυασμό των προαναφερθέντων. Επίσης παρέχεται η δυνατότητα ρύθμισης ενός φίλτρου πρόσβασης (access filter) ο οποίος καθορίζει ποιού τύπου μεθόδους και μεταβλητών μπορεί να χειριστεί ο γράφος. Ο γράφος δεν είναι στατικός, προσδίδοντας στον χρήστη τη δυνατότητα να περιηγηθεί σε χαμηλότερου επιπέδου πληροφορίες τις οποίες έχει συλλέξει ο γράφος. Η περιήγηση αυτή επιτυγχάνεται γραφικά.

Επιπλέον η πλατφόρμα παρέχει τρεις τύπους αναλυτών οι οποίοι μπορούν να εντοπίσουν κυκλικές αναφορές, κρίσιμα σημεία (hubs), καθώς και κόμβους προς τους οποίους δεν υπάρχει κάποια αναφορά. Οι κυκλικές αναφορές πρέπει να αποφεύγονται έτσι ώστε να μειωθεί η παρουσία αδιεξόδων (deadlocks). Πρέπει να γνωρίζουμε τα κρίσιμα σημεία για να ξέρουμε ποια κομμάτια κώδικα είναι πιο επιρρεπή σε σφάλματα. Επίσης σημεία στα οποία κάθε αλλαγή είναι επίφοβη λόγο των αρκετών εξαρτήσεων που υπάρχουν προς και από αυτό.

- Διάγραμμα Επιπέδων (Layer Diagram)

Όταν ο χρήστης καθορίσει τα επίπεδα του υπό ανάπτυξη ή στη φάση ελέγχου συστήματος και πιο συγκεκριμένα του «solution», υπάρχει η δυνατότητα παρουσίασης της συσχέτισης μεταξύ των υπομονάδων (modules) και των έργων (projects) που βρίσκονται στο συγκεκριμένο «solution». Η συσχέτιση αυτή αναφέρεται ως προς τα επίπεδα που έχουν καθοριστεί από τον χρήστη. Έτσι ο χρήστης μπορεί να επιβεβαιώσει εάν η αρχιτεκτονική του συστήματός του είναι η ορθή ή όχι. Ουσιαστικά την επιβεβαίωση αυτή την κάνει η πλατφόρμα για αυτόν. Ελέγχει εάν ο κώδικας ακολουθεί τις προηγούμενες συσχετίσεις που έχουν γίνει

και εάν δεν ισχύουν τότε παρουσιάζονται στον χρήστη τα σχετικά μηνύματα καθώς και υποδείξεις που παραβιάζονται οι σχέσεις.

- Όλα τα UML διαγράμματα
    - Class Diagram
    - Sequence Diagram
    - Activity Diagram
    - Component Diagram

Η πλατφόρμα Visual Studio 2010, παίρνει σαν είσοδο το λογισμικό και βάσει του πλαισίου εργασίας αρχιτεκτονικής το αναλύει και παράγει τα πιο πάνω διαγράμματα

3.10 Εξωτερικά Εργαλεία Ελέγχου (External Testing Tools)

Η ευρεία χρήση της πλατφόρμας, οδήγησε αρκετούς ερευνητές καθώς και εταιρίες ανάπτυξης λογισμικών συστημάτων να αναπτύξουν δικά τους εργαλεία τα οποία στη συνέχεια χρησιμοποίησαν αρκετοί προγραμματιστές ανά το παγκόσμιο. Πιο κάτω θα παρουσιασθούν δύο από τα πιο γνωστά εργαλεία τα οποία βοηθούν τη διαδικασία ελέγχου ενός λογισμικού.

Το πρώτο εργαλείο είναι το PEX [16], το οποίο προήλθε από την ερευνητική ομάδα του Visual Studio. Σκοπός του εργαλείου αυτού είναι η αυτόματη παραγωγή περιπτώσεων δοκιμής με την τεχνική ελέγχου άσπρου κουτιού. Ο χρήστης μπορεί να φυλάξει ένα σύνολο από αυτές τις περιπτώσεις δοκιμής (test cases) σε μία συλλογή ελέγχων (test suite) το οποίο θα στοχεύει σε υψηλά ποσοστά κάλυψης κώδικα. Το PEX [16] χρησιμοποιεί ένα άλλο υπό-σύστημα το Moles [16], το οποίο στοχεύει στην απομόνωση μεθόδων του κώδικα με τεχνικές λοξοδρόμησης και αποκοπής.

Το δεύτερο εργαλείο είναι το Altova UModel [17], το οποίο δημιουργήθηκε από την εταιρία Altova για να προσδίδει την δυνατότητα στην πλατφόρμα να δημιουργεί UML

διαγράμματα βάσει του πηγαίου κώδικα χρησιμοποιώντας reverse engineering τεχνικές. Μερικά από τα διαγράμματα που μπορεί να δημιουργήσει είναι

- Use Case Diagram
- Class Diagram
- Activity Diagram
- Composite Structure Diagram
- Deployment Diagram
- Interaction Overview
- Object Diagram
- Package Diagram
- Sequence Diagram
- State Machine Diagram
- XML Schema as UML diagram
- Timing Diagram
- Profile Diagram
- Business Process (BPMN) Diagram

Υπάρχουν και άλλα εργαλεία τα οποία θα μπορούσαν να συνδυαστούν μεταξύ τους έτσι ώστε να δημιουργήσουν υβριδικά εργαλεία με πολλαπλές δυνατότητες ως προς τον έλεγχο λογισμικών συστημάτων σε όλες τις προσφερόμενες γλώσσες προγραμματισμού από την πλατφόρμα Visual Studio. Μερικά τέτοια εργαλεία είναι :

- NUnit [18]
- Gallio [19]
- NCover [20]

Λόγω των πιο πάνω δυνατοτήτων και προοπτικών λήφθηκε η απόφαση μεταφοράς του ερευνητικού ενδιαφέροντος από την πλατφόρμα Java στην πλατφόρμα Visual Studio η οποία όπως κάποιος μπορεί εύκολα να αντιληφθεί παρέχει ένα αρκετά μεγάλο σύνολο από δυνατότητες ειδικά στον τομέα ελέγχου συστημάτων λογισμικού. Αρκετές προηγούμενες έρευνες μπορούν να υλοποιηθούν στην πλατφόρμα Visual Studio και να χρησιμοποιηθούν για επόμενες μελέτες. Για παράδειγμα, η δημιουργία Γράφου Ροής Ελέγχου, Γράφου Εξαρτήσεων, χρήση γενετικών αλγορίθμων για σκοπούς ελέγχου συστημάτων λογισμικού, και γενικότερα το πλαίσιο εργασίας ελέγχων λογισμικών προγραμμάτων γραμμένα σε Java. Η μεταφορά αυτή θα δώσει νέες προοπτικές δημιουργίας υβριδικών μεθόδων ελέγχου συστημάτων λογισμικού συνδυάζοντας είδη

υπάρχον εργαλεία/αλγορίθμους καθώς και νέες υλοποιήσεις. Επίσης η ευρεία χρήση της πλατφόρμας αυτής σε αντίθεση με αυτή της Java θεωρείται ένα επιπλέον κίνητρο κυρίως λόγο της ανάγκης που δημιουργείται για την αυτοματοποίηση της διαδικασίας ελέγχου συστημάτων λογισμικού.

Κεφάλαιο 4

Έλεγχος βάσει μεταλλάξεων πηγαίου κώδικα (Mutation Testing) και εργαλεία παραγωγής μεταλλάξεων πηγαίου κώδικα (Mutation Engine Tools)

4.1 Εισαγωγή

Μία τεχνική ελέγχου της μορφής άσπρου κουτιού (white-box testing), είναι αυτή του ελέγχου βάσει μεταλλάξεων πηγαίου κώδικα (mutation testing). Η τεχνική αυτή, την οποία εισήγαγαν οι Hamlet [2] και DeMillo [3], βασίζεται στην εισαγωγή λαθών σε ένα πρόγραμμα. Η εισαγωγή λαθών σε ένα λογισμικό σύστημα, βασίζεται στην μετάλλαξή του. Η μετάλλαξη πηγαίου κώδικα είναι μια τεχνική με την οποία μεταλλάσσεται ο πηγαίος κώδικας βάσει κάποιων τελεστών μετάλλαξης χωρίς όμως να παραβιάζονται οι περιορισμοί της γραμματικής της γλώσσας, τόσο σημασιολογικά όσο και συντακτικά. Η μετάλλαξη του κώδικα είναι ατομική έτσι ώστε να μην απέχει κατά πολύ από το αρχικό λογισμικό.

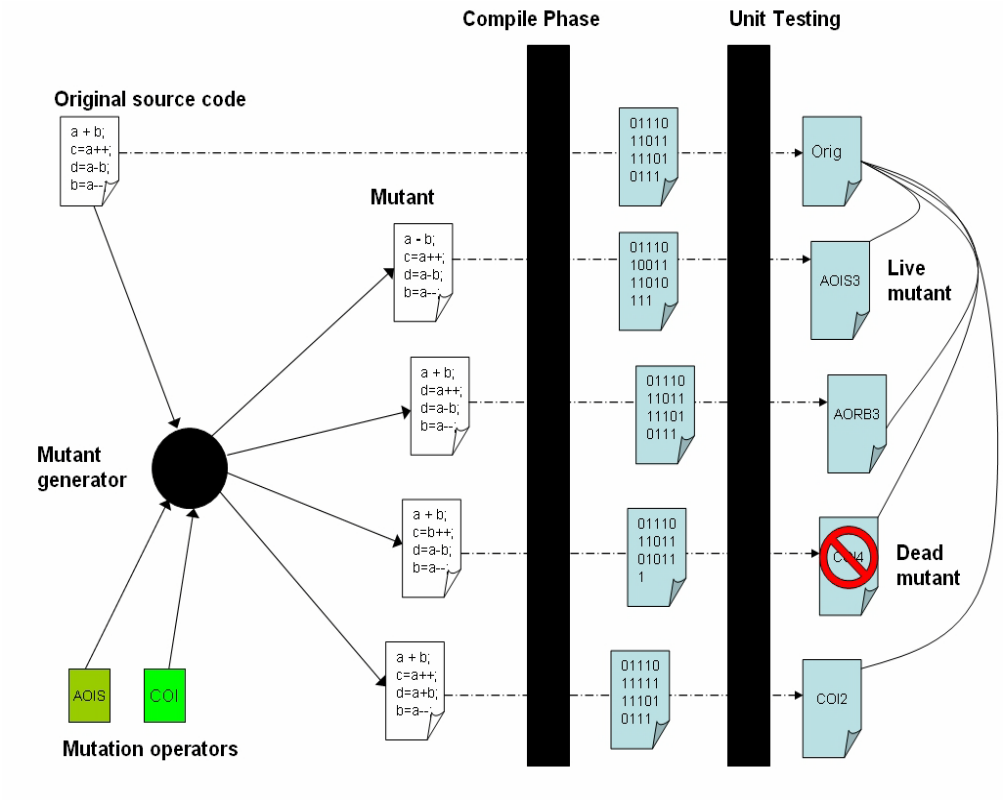
Κάποιος θα αναρωτηθεί όμως, πώς η εισαγωγή ενός σφάλματος μπορεί να βοηθήσει στη διαδικασία ελέγχου του. Η τεχνική ελέγχου βάσει μεταλλάξεων πηγαίου κώδικα (mutation testing), δεν είναι μια στρατηγική ελέγχου όπως για παράδειγμα ο έλεγχος δεδομένων ή/και μονοπατιών (data and path testing). Δεν περιγράφει κριτήρια επιλογής δεδομένων ελέγχου. Στόχος της πιο πάνω τεχνικής είναι να επιβεβαιώσει ή να διαψεύσει την καταλληλότητα των περιπτώσεων δοκιμής. Μπορεί να χρησιμοποιηθεί σε συνδυασμό με άλλες τεχνικές έτσι ώστε να ενισχύσει την επιλογή των περιπτώσεων δοκιμής (test cases) κάνοντας επιβεβαίωση της καταλληλότητάς τους, αλλά και με

άλλες τεχνικές οι οποίες προορίζονται για εύρεση σφαλμάτων σε ένα λογισμικό σύστημα.

Το σφάλμα το οποίο προστίθεται στον κώδικα, αντιπροσωπεύει συχνά σφάλματα τα οποία παρατηρούνται από τους προγραμματιστές. Οι προγραμματιστές τείνουν να δημιουργούν λογισμικά συστήματα όσο το δυνατό πιο κοντά στην ορθή λειτουργία του. Από αυτό συμπεραίνουμε ότι συνήθως τα πιθανά σφάλματα μπορούν να ανευρεθούν μέσω των μεταλλάξεων που μας παρέχει η τεχνική ελέγχου βάσει μεταλλάξεων του πηγαίου κώδικα (mutation testing).

4.2 Έλεγχος βάσει μεταλλάξεων πηγαίου κώδικα

Η τεχνική ελέγχου βάσει μεταλλάξεων πηγαίου κώδικα (Mutation Testing) [6] είναι μία διαδικασία η οποία αποσκοπεί στο να βοηθήσει στον έλεγχο ενός λογισμικού συστήματος. Το πιο κάτω σχήμα παρουσιάζει πως εφαρμόζεται ο αλγόριθμος. Δοθέντος ενός προγράμματος, με την βοήθεια μιας μηχανής παραγωγής μεταλλάξεων πηγαίου κώδικα (mutation engine), παράγονται ένα σύνολο από προγράμματα τα οποία διαφέρουν στο ελάχιστο από τον αρχικό λόγω της μετάλλαξης που έχει γίνει στο κάθε ένα από αυτό βάσει ενός συνόλου τελεστών μετάλλαξης. Τελεστές μετάλλαξης είναι οι κανόνες τροποποίησης ενός πηγαίου κώδικα, με το να προσθαφαιρούν ή ακόμη και να τροποποιούν σύμβολα, μεταβλητές και εκφράσεις που εμφανίζονται στον συγκεκριμένο κώδικα.



Σχήμα 4.1 – Αλγόριθμος mutation testing [6]

Ακολούθως, γίνεται η επιβεβαίωση της ορθότητας του αρχικού λογισμικού βάσει ενός συνόλου περιπτώσεων δοκιμής. Εάν δεν επιβεβαιώνεται αυτό, τότε το λογισμικό πρέπει να διορθωθεί έτσι ώστε να συνεχισθεί η διαδικασία. Μετά το βήμα αυτό, κάθε μεταλλαγμένο πρόγραμμα θα εκτελεστεί με το ίδιο σύνολο περιπτώσεων δοκιμής (test cases) που έχουν χρησιμοποιηθεί προηγουμένως και έχουν εξασφαλίσει την ορθότητά του. Εάν το σύνολο των περιπτώσεων δοκιμής οδηγήσει σε διαφορετικά αποτελέσματα σε σχέση με το αρχικό, τότε το μεταλλαγμένο λογισμικό «σκοτώνεται» αλλιώς «επιβιώνει». Στόχος του αλγορίθμου είναι να οδηγήσει το μεταλλαγμένο λογισμικό σε αποτυχία. Ο βαθμός μετάλλαξης (αναλογία «σκοτωμένων» μεταλλαγμένων λογισμικών σε σχέση με αυτά που έχουν «επιβιώσει») για κάθε σύνολο από περιπτώσεις δοκιμής (test cases) υποδηλώνει την καταλληλότητα του συνόλου των περιπτώσεων δοκιμής. Μέγιστος βαθμός μετάλλαξης, ισοδυναμεί με επιτυχία εύρεσης όλων των σφαλμάτων τα οποία ενσωματώθηκαν στον αρχικό κώδικα άρα το σύνολο των περιπτώσεων δοκιμής κρίνεται ως επιτυχές και ικανοποιητικό.

Παράδειγμα αποτίμησης βαθμού καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής

Αρχικό πρόγραμμα:

```
r:= 1;
For i:= 2 to 3 do
if a[i] > a[r] then r:=i;
```

Μεταλλαγμένα προγράμματα:

M1

```
r:= 1;
For i:= 1 to 3 do
if a[i] > a[r] then r:=i;
```

M2

```
r:= 1;
For i:= 2 to 3 do
if i > a[r] then r:=i;
```

M3

```
r:= 1;
For i:= 2 to 3 do
if a[i] >= a[r] then r:=i;
```

M4

```
r:= 1;
For i:= 2 to 3 do
if a[r] > a[r] then r:=i;
```

Περιπτώσεις Δοκιμής:

	a[1]	a[2]	a[3]
TC1	1	2	3
TC2	1	2	1
TC3	3	1	2

Αποτελέσματα μετά από την εκτέλεση της μεθόδου ελέγχου με χρήση μεταλλάξεων

	Επιθυμητό αποτέλεσμα (r)	M1	M2	M3	M4	«σκοτωμένα» προγράμματα
TC1	3	3	3	3	1	M4
TC2	2	2	3	2	1	M2 & M4
TC3	1	1	1	1	1	--

Από τα πιο πάνω αποτελέσματα, βλέπουμε ότι τα μεταλλαγμένα προγράμματα M1 και M3 δεν έχουν αναγνωριστεί από το συγκεκριμένο σύνολο περιπτώσεων δοκιμής, με αποτέλεσμα να κρίνετε μη κατάλληλο. Από αυτό συμπεραίνουμε πως το συγκεκριμένο σύνολο περιπτώσεων δοκιμής χρειάζεται ενίσχυση με νέες περιπτώσεις δοκιμής έτσι ώστε να καταστεί κατάλληλο για να αναγνωρίζει όλα τα εμφυτευμένα σφάλματα.

Αρκετές φορές όμως υπάρχουν ισότιμα μεταλλαγμένα προγράμματα ως προς το αρχικό, τα οποία «επιβιώνουν» μέχρι το τέλος της διαδικασίας αυτής. Αυτό χαρακτηρίζεται ως το μεγαλύτερο και άλυτο πρόβλημα που υπάρχει στην τεχνική ελέγχου βάσει μεταλλάξεων (mutation testing).

Παράδειγμα Ισοδύναμων μεταλλάξεων:

Αρχικό πρόγραμμα

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

Μετά από μετάλλαξη, συγκεκριμένα με σχεσιακό τελεστή έχουμε το πιο κάτω πρόγραμμα.

Μεταλλαγμένο πρόγραμμα

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

Εύκολα κάποιος μπορεί να παρατηρήσει πως η σχέση ισότητας παραμένει. Αποτέλεσμα αυτού η ύπαρξη μιας περίπτωσης δοκιμής (test case) η οποία επιβεβαιώνει και τα δύο προγράμματα χωρίς να εμφανίζεται οποιαδήποτε ένδειξη διαφορετικότητας.

Επίσης η μεγάλη υπολογιστική δύναμη που χρειάζεται και ο μεγάλος αριθμός των μεταλλαγμένων προγραμμάτων είναι άλλο ένα σοβαρό πρόβλημα το οποίο μειώνει την αποδοτικότητα του αλγορίθμου.

Μία περίπτωση δοκιμής (test case) είναι κατάλληλη εάν και μόνο αν είναι σε θέση να εντοπίσει την ύπαρξη σφαλμάτων στο αρχικό πρόγραμμα. Με την τεχνική ελέγχου βάσει μεταλλάξεων πηγαίου κώδικα (mutation testing) επιτυγχάνουμε μέγιστο βαθμό καταλληλότητας των test cases πράγμα πολύ ενθαρρυντικό. Ουσιαστικά απαλείφουμε την ύπαρξη ακατάλληλων περιπτώσεων δοκιμής (test cases). Ένα σύνολο από κατάλληλες περιπτώσεις δοκιμής (test cases) είναι ένα δυνατό χαρτί για ένα προγραμματιστή, ο οποίος μπορεί να το αξιοποιήσει για διάφορες μετρικές του κώδικα καθώς και για μελλοντική συντήρηση του συγκεκριμένου λογισμικού.

Επίσης η τεχνική αυτή μπορεί να συνδυαστεί με άλλες τεχνικές ως μία υβριδική λύση για καλύτερο έλεγχο προγραμμάτων καθώς και για εύρεση σφαλμάτων για την αποσφαλμάτωση λογισμικών συστημάτων.

4.3 Τελεστές μετάλλαξης πηγαίου κώδικα

Ο τελεστής μετάλλαξης είναι υπεύθυνος για την μετάλλαξη του πηγαίου κώδικα. Με λίγα λόγια είναι οι κανόνες βάσει της γραμματικής της κάθε γλώσσας προγραμματισμού με τους οποίους ένα ήδη υπάρχον λογισμικό μπορεί να μεταλλαχθεί και να διατηρεί την ακεραιότητά του.

Υπάρχουν αρκετών ειδών τελεστές μετάλλαξης, οι οποίοι διαφέρουν σχετικά με κάθε γλώσσα προγραμματισμού, αλλά χωρίζονται κυρίως σε δύο κατηγορίες. Τους τελεστές που αναφέρονται σε αλλαγές σε επίπεδο μεθόδων, και τους τελεστές που αναφέρονται σε αλλαγές στην κλάση αλλά και μεταξύ κλάσεων. Σε αυτή την εργασία ασχοληθήκαμε μόνο με της πρώτης κατηγορίας τελεστές.

Το εργαλείο μετάλλαξης MuJava [22], το οποίο θεωρείται ως ένα από τα αξιόπιστα ερευνητικά αλλά και στην αγορά, χρησιμοποιεί τους πιο κάτω τελεστές όπως εμφανίζονται στους πιο κάτω πίνακες. Δεν θα γίνει η λεπτομερής ανάλυσή τους γιατί σε επόμενο κεφάλαιο θα παρουσιαστούν λεπτομερώς οι τελεστές τους οποίους χρησιμοποίησε το υλοποιημένο εργαλείο μετάλλαξης κώδικα.

Operator	Description
AOR	Arithmetic Operator Replacement
AOI	Arithmetic Operator Insertion
AOD	Arithmetic Operator Deletion
ROR	Relational Operator Replacement
COR	Conditional Operator Replacement
COI	Conditional Operator Insertion
COD	Conditional Operator Deletion
SOR	Shift Operator Replacement
LOR	Logical Operator Replacement
LOI	Logical Operator Insertion
LOD	Logical Operator Deletion
ASR	Assignment Operator Replacement

Σχήμα 4.2 – Τελεστές μετάλλαξης – Επίπεδο μεθόδου

Language Feature	Operator	Description
Encapsulation	AMC	**A**ccess modifier **c**hange
Inheritance	IHD	**H**iding variable **d**eletion
	IHI	**H**iding variable **i**nsertion
	IOD	**O**verriding method **d**eletion
	IOP	**O**verriding method calling **p**osition change
	IOR	**O**verriding method **r**ename
	ISI	**super** keyword **i**nsertion
	ISD	**super** keyword **d**eletion
	IPC	Explicit call to a **p**arent's **c**onstructor deletion
Polymorphism	PNC	**n**ew method call with **c**hild class type
	PMD	**M**ember variable **d**eclaration with parent class type
	PPD	**P**arameter variable **d**eclaration with child class type
	PCI	Type **c**ast operator **i**nsertion
	PCC	**C**ast type **c**hange
	PCD	Type **c**ast operator **d**eletion
	PRV	**R**eference assignment with other comparable **v**ariable
	OMR	**O**verloading **m**ethod contents **r**eplace
	OMD	**O**verloading **m**ethod **d**eletion
	OAC	**A**rguments of **o**verloading method call **c**hange
Java-Specific Features	JTI	**this** keyword **i**nsertion
	JTD	**this** keyword **d**eletion
	JSI	**static** modifier **i**nsertion
	JSD	**static** modifier **d**eletion
	JID	Member variable **i**nitialization **d**eletion
	JDC	Java-supported **d**efault constructor **c**reation
	EOA	**R**eference **a**ssignment and content assignment replacement
	EOC	**R**eference **c**omparison and content comparison replacement
	EAM	**A**cessor **m**ethod change
	EMM	**M**odifier **m**ethod change

Σχήμα 4.2 – Τελεστές μετάλλαξης – Επίπεδο κλάσης

4.4 Μηχανή παραγωγής μεταλλάξεων πηγαίου κώδικα

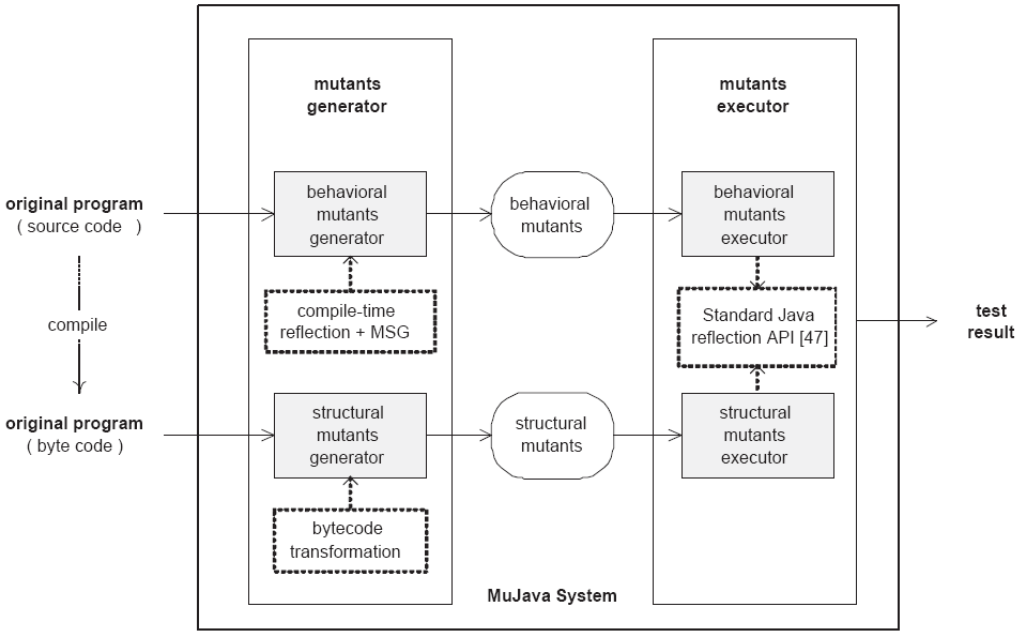
Η μηχανή παραγωγής μεταλλάξεων πηγαίου κώδικα είναι το κύριο συστατικό της τεχνικής ελέγχου βάσει μεταλλάξεων (mutation testing). Στόχος της μηχανής παραγωγής μεταλλάξεων είναι η παραγωγή μεταλλαγμένων λογισμικών βάσει του λογισμικού υπό έλεγχο. Οι μεταλλάξεις γίνονται με γνώμονα ένα σύνολο από διαθέσιμους τελεστές μετάλλαξης. Όλες οι αλλαγές δεν αλλάζουν την εγκυρότητα του λογισμικού ως μία εκτελέσιμη οντότητα, αλλά γίνεται μία παραλλαγή της λειτουργίας καθώς και της συμπεριφοράς του λογισμικού.

Έχουν προταθεί αρκετά εργαλεία τα οποία χρησιμοποιούν μία μηχανή παραγωγής μεταλλάξεων έτσι ώστε να πετύχουν την τεχνική ελέγχου βάσει μεταλλάξεων (mutation testing), γνωστά ως εργαλεία ελέγχου βάσει μεταλλάξεων (mutation testing tools). Σε αυτό το υποκεφάλαιο θα παρουσιαστεί ένα σύνολο από αυτού του είδους τα εργαλεία τα οποία αναφέρονται σε αρκετές γλώσσες προγραμματισμού.

Το εργαλείο Mothra [21], είναι το πρώτο ολοκληρωμένο σύστημα ελέγχου με τη χρήση μετάλλαξης. Απευθύνεται στη γλώσσα προγραμματισμού Fortran 77 και παρέχει ένα ευέλικτο και πλήρες περιβάλλον ελέγχου. Το Mothra [21] αλλάζει ένα ενδιάμεσο κώδικα που σχεδιάζεται ειδικά για τις μεταλλάξεις.

Το εργαλείο MuJava [22] είναι άλλο εργαλείο ελέγχου με τη χρήση μετάλλαξης το οποίο απευθύνεται στην αντικειμενοστρεφή γλώσσα προγραμματισμού Java. Θεωρείται ως ένα από τα πιο ολοκληρωμένα συστήματα αναφορικά με τη γλώσσα προγραμματισμού Java. Υποστηρίζει ένα μεγάλο φάσμα από τελεστές μετάλλαξης. Οι τελεστές μετάλλαξης αναφέρονται τόσο στον παραδοσιακό έλεγχο όσο και σε έλεγχο στο επίπεδο κλάσεων. Χρησιμοποιεί την υπάρχουσα μέθοδο MSG για την παραγωγή «meta-mutant» προγράμματος στο επίπεδο source, στο οποίο ενσωματώνει πολλές μεταλλάξεις. Εργάζεται σε επίπεδο bytecode, πράγμα που του δίνει την δυνατότητα να είναι αρκετά γρήγορο και να χρειάζεται μόνο δύο μεταγλωττίσεις, μια του αρχικού προγράμματος και μία του «meta-mutant», σε αντίθεση με άλλα εργαλεία τα οποία μεταγλωττίζουν όλα τα μεταλλαγμένα προγράμματα. Υπάρχει και το εργαλείο MuClipse [23] το οποίο είναι μια παραλλαγή του MuJava [22] έτσι ώστε να

χρησιμοποιείται ως add-in στην πλατφόρμα Eclipse. Στο πιο κάτω σχήμα παρατηρούμε την αρχιτεκτονική του εργαλείου αυτού.



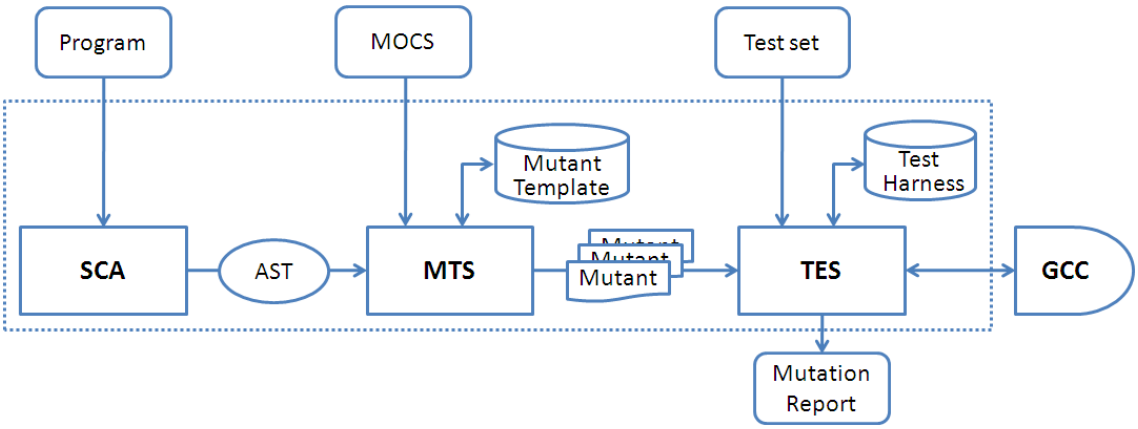
Σχήμα 4.3 - Αρχιτεκτονική εργαλείου Mu-Java

Το εργαλείο Jester [24] το οποίο είναι open-source και αποσκοπεί στον έλεγχο Java προγραμμάτων με τη χρήση μετάλλαξης. Είναι βασισμένο στο framework JUnit το οποίο ασκεί έλεγχο σε ένα κομμάτι του κώδικα δημιουργώντας test cases. Χρησιμοποιεί απλούς τελεστές μετάλλαξης. Μεταγλωττίζει κάθε τελεστή μετάλλαξης πράγμα που του μειώνει την αποδοτικότητα. Επίσης παρατηρήθηκαν περιστατικά στα οποία το Jester [24] έκανε μεταλλάξεις σε σχόλια και όχι στον κώδικα. Λόγω των περιορισμένων τελεστών μετάλλαξης δεν υπερτερεί κατά πολύ από τα εργαλεία ποσοστού κάλυψης πηγαίου κώδικα (code coverage tools). Επίσης δεν είναι αποδοτικό αλλά ούτε και αξιόπιστο για μεγάλου μεγέθους προγράμματα.

Το εργαλείο Nester [25] είναι open-source και αποσκοπεί στον έλεγχο C# προγραμμάτων με σκοπό την αποτίμηση της καταλληλότητας των Unit Tests. Χρησιμοποιεί μεταλλαγμένα προγράμματα για να ελέγξει εάν υφιστάμενα test cases μπορούν να διαχωρίσουν το αρχικό πρόγραμμα από τα μεταλλαγμένα. Υποστηρίζει μόνο προγράμματα τα οποία είναι γραμμένα στην πλατφόρμα Visual Studio 2005 και

υποστηρίζει, τουλάχιστον μέχρι τη σημερινή έκδοση, μόνο το NUnit framework. Το Nester χρωματίζει τον κώδικα βάσει της κατάστασής του μετά από την ανάλυσή του. Με πράσινο φόντο παρουσιάζονται τα σημεία όπου επιτυχώς μετά το mutation testing το Nester πέτυχε να διαχωρίσει το αρχικό από το μεταλλαγμένο λογισμικό. Με κόκκινο φόντο παρουσιάζονται τα σημεία όπου δεν έχουν αναγνωριστεί από το Jester [24], καθώς μετά την μετάλλαξη τα αποτελέσματα των Unit Tests παραμένουν ως έχουν. Τέλος με μπλε φόντο παρουσιάζονται τα κομμάτια κώδικα τα οποία δεν έχουν επισκεφθεί καθόλου από τα Unit Tests. Το Nester [25] χρησιμοποιεί περιορισμένο αριθμό μεταλλάξεων, όπως αριθμητικές, λογικές και σχεσιακές. Μιας και το Nester [25] είναι ακόμη σε αρχικά στάδια, υπάρχουν αρκετά bugs τα οποία όμως η ομάδα σε συνεργασία με τους χρήστες προσπαθούν να το φέρουν σε μία σταθερή κατάσταση.

Το εργαλείο MILU [26] είναι ένα εργαλείο ελέγχου λογισμικού με τη χρήση μετάλλαξης το οποίο προορίζεται για τον έλεγχο λογισμικών συστημάτων της γλώσσας προγραμματισμού C. Είναι σχεδιασμένο να εφαρμόζει «higher order mutation testing» και κατ'επέκταση «first order mutation testing». Με τον ορισμό «first order mutation testing» αναφερόμαστε στις μεταλλάξεις που χρησιμοποιούν όλα τα πιο πάνω εργαλεία, δηλαδή ατομικές μεταλλάξεις, που εμφανίζονται μόνο μια φορά σε κάθε μεταλλαγμένο λογισμικό. Αντιθέτως, με τον ορισμό «higher order mutation testing» αναφερόμαστε στην εφαρμογή περισσότερων της μίας μετάλλαξης ανά μεταλλαγμένο λογισμικό. Επίσης το εργαλείο αυτό δίδει τη δυνατότητα στον χρήστη να καθορίσει δικούς του τελεστές μετάλλαξης. Στόχος του εργαλείου είναι να ξεχωρίσει το αρχικό πρόγραμμα από τα μεταλλαγμένα βάσει ενός συνόλου από περιπτώσεις δοκιμής. Έτσι θα εγγυηθεί πότε ένα σύνολο από περιπτώσεις δοκιμής είναι κατάλληλο (βαθμός καταλληλότητας) ή ακατάλληλο. Ακόμη παρέχει δικό του API, παρέχοντας τη δυνατότητα στους ενδιαφερόμενους ερευνητές να μελετήσουν το χώρο της τεχνικής ελέγχου βάσει μεταλλάξεων (mutation testing) με τη βοήθεια του εργαλείου αυτού. Οι ενδιαφερόμενοι ερευνητές μπορούν να καθορίσουν νέους τελεστές μετάλλαξης καθώς και νέα «fitness function» βάσει των δικών τους ιδεών και στόχων. Το πιο κάτω σχήμα παρουσιάζει την αρχιτεκτονική του εργαλείου αυτού.



Σχήμα 4.4 Αρχιτεκτονική εργαλείου MILU

Το κομμάτι SCA λειτουργεί ως ένας C parser ο οποίος τεμαχίζει τον κώδικα που δέχεται ως είσοδο και κρατά χρήσιμες πληροφορίες σε λίστες. Ακόμη δημιουργεί ένα απλό Abstract Source Tree το οποίο στέλνει σαν είσοδο στο επόμενο κομμάτι που είναι το MTS. Το κομμάτι MTS, είναι υπεύθυνο να εκτελέσει τις μεταλλάξεις που έχει επιλέξει ο χρήστης δημιουργώντας ένα «mutant template» όπου φυλάγει τις πιθανές αλλαγές και τους τύπους τους. Το κομμάτι TES χρησιμοποιείται κυρίως για να εκτελέσει τα μεταλλαγμένα προγράμματα τα οποία παίρνει σαν είσοδο από το κομμάτι MTS εφαρμόζοντας ένα σύνολο από περιπτώσεις δοκιμής. Η εκτέλεση από το κομμάτι GCC δεν πραγματοποιείται από μεταγλωττιστή αλλά από μια κοινή βιβλιοθήκη μέσω του Test Harness για λόγους αποδοτικότητας.

Κεφάλαιο 5

Παρουσίαση των συστατικών για την πλατφόρμα Visual Studio, της μηχανής παραγωγής μετάλλαξης εντολών σε πηγαίο κώδικα και της καινοτόμου ιδέας «βελτιστοποίησης» της πιο πάνω μηχανής.

5.1 Εισαγωγή

Με την ανάλυση των προηγούμενων κεφαλαίων, παρατηρούμε τη δυναμική της πλατφόρμας Visual Studio 2010, καθώς και της πληθώρας των δυνατοτήτων που παρέχει στον τομέα ελέγχου λογισμικού συστήματος. Επιπλέον η δυναμική που αποκτά η πλατφόρμα είναι αυτή των πολλών εφαρμογών, open-source ή μη, που εμφανίζονται από διάφορες εταιρίες καθώς και ερευνητές.

Επίσης, παρατηρούμε την σημαντικότητα μιας μηχανής παραγωγή μεταλλάξεων τόσο στην τεχνική εκτίμησης καταλληλότητας περιπτώσεων δοκιμής, ελέγχου βάσει μεταλλάξεων, καθώς και σε άλλους τομείς της διαδικασίας ελέγχου ενός λογισμικού συστήματος, όπως για παράδειγμα την εύρεση σφαλμάτων.

Οι δύο πιο πάνω λόγοι, οδήγησαν στην εστίαση του ερευνητικού ενδιαφέροντος προς αυτές τις δύο κατευθύνσεις. Αποτέλεσμα αυτού η υλοποίηση δύο ανεξάρτητων κομματιών (components) τα οποία θα δίδουν τη δυνατότητα της επικύρωσης προγραμμάτων γραμμένα στην πλατφόρμα Visual Studio 2010 καθώς και τη δυνατότητα λεπτομερούς ανάλυσης, χωρίς εκτέλεση, του πηγαίου κώδικα το οποίο είναι γραμμένο στη γλώσσα προγραμματισμού C#. Επιπλέον, υλοποιήθηκε μία μηχανή

παραγωγής μεταλλάξεων (σε επίπεδο μεθόδων) πηγαίου κώδικα με τη χρήση των πιο πάνω κομματιών. Στα επόμενα υπό-κεφάλαια θα παρουσιαστεί η ανάλυση της σχετικής δουλειάς που έχει γίνει σε αυτή τη διπλωματική εργασία.

5.2 Συστατικά Επικύρωσης και Στατικής Ανάλυσης Πηγαίου Κώδικα

Μιας και η πλατφόρμα Visual Studio 2010 είναι μια σχετικά νέα έκδοση, απέχουμε μόνο ελάχιστους μήνες μετά την δημοσίευσή της, κρίθηκε αναγκαίο να υλοποιήσουμε κάποια επαναχρησιμοποιήσιμα συστατικά (components) τα οποία θα αποτελούν τη ραχοκοκαλιά για επόμενες μελέτες οι οποίες θα αναφέρονται στην πλατφόρμα αυτή. Τα πιο κάτω συστατικά δεν υπάρχουν υλοποιημένα για το πλαίσιο εργασίας «.Net framework 4.0». Αρκετοί χρησιμοποιούν παλαιότερα εργαλεία αντιμετωπίζοντας όμως προβλήματα συμβατότητας.

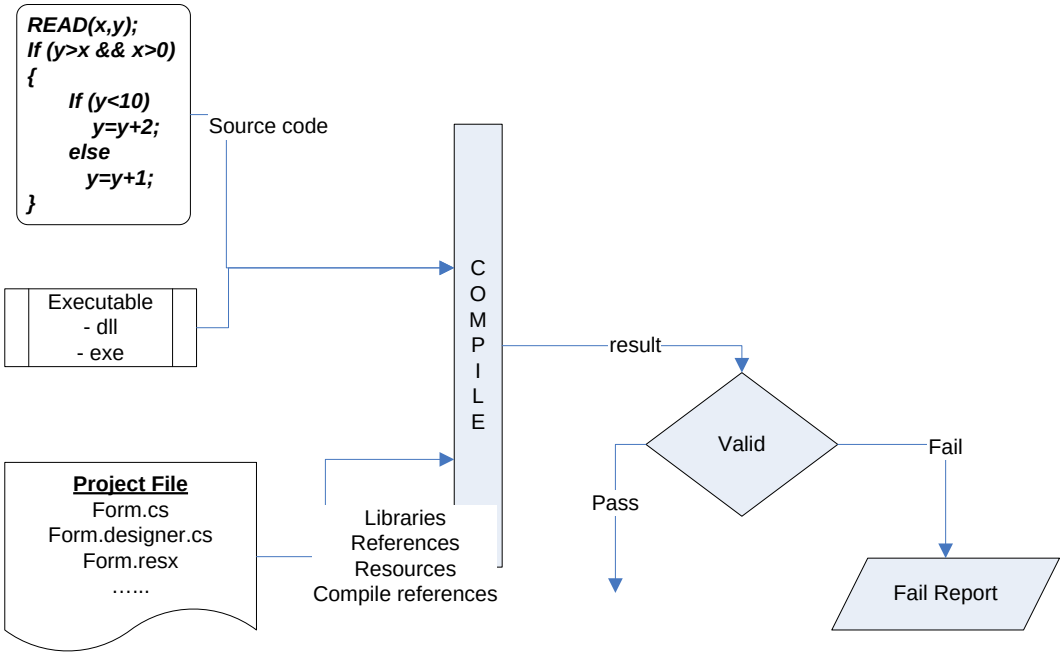
5.2.1 Επικύρωση Πηγαίου Κώδικα

Το πρώτο συστατικό το οποίο έχει υλοποιηθεί είναι αυτό της επικύρωσης πηγαίου κώδικα. Σκοπός του, είναι η μεταγλώττιση του κώδικα και η εμφάνιση των λαθών που υπάρχουν στον κώδικα εάν και εφόσον δεν περάσει την φάση επιβεβαίωσης. Δηλαδή εντοπίζει εάν υπάρχουν συντακτικά λάθη στον πηγαίο κώδικα. Το συστατικό αυτό, δέχεται ως είσοδο τον πηγαίο κώδικα (.cs) ή το εκτελέσιμο αρχείο (.exe) ή την εκτελέσιμη βιβλιοθήκη (.dll) καθώς και το project file (.csproj). Ο λόγος που χρειάζεται το project file είναι για να υπάρχει η δυνατότητα εύρεσης των αναφορών (references) σε εκτελέσιμα αρχεία (βιβλιοθήκες ή άλλα dll αρχεία) που υπάρχουν μέσω της εντολής «using» αλλά και για αναφορές σε αρχεία που αποτελούν μέρος της εφαρμογής (για παράδειγμα το design αρχείο μιας window εφαρμογής). Το project file μπορεί να διαβαστεί σαν ένα xml αρχείο. Έτσι το συστατικό αυτό είναι ενισχυμένο με τη δυνατότητα ανάλυσης xml αρχείων.

Κατά την μεταγλώττιση ενός πηγαίου κώδικα στη γλώσσα προγραμματισμού C#, πρέπει να συμπεριληφθούν και τα μονοπάτια ως προς τις αναφορές των εξωτερικών εκτελέσιμων αρχείων. Έτσι το συστατικό εκτός από την εύρεση των ονομάτων των αρχείων αυτών πρέπει να ψάξει για το πού βρίσκονται στον υπολογιστή που τρέχει το λογισμικό. Συνήθως τα εκτελέσιμα αρχεία βρίσκονται μέσα στον φάκελο όπου βρίσκεται το λογισμικό και πιο συγκεκριμένα στο αρχείο bin (κάποιος μπορεί να το

επιλέξει να γίνεται αυτό μέσα από τα properties του project/solution). Σε αρκετές περιπτώσεις όμως, τα αρχεία αυτά βρίσκονται στο «global assembly cash» του οποίου το μονοπάτι είναι το “C:/WINDOWS/assembly/”. Τέλος εάν δεν βρίσκονται σε κανένα από τα πιο πάνω μονοπάτια, το συστατικό ψάχνει την ύπαρξή τους στο μονοπάτι “C:/Program Files/Reference Assemblies/Microsoft/Framework/.NETFramework/v4.0/Profile/Client/".

Το συστατικό χρησιμοποιεί την κλάση CodeDomProvider για να χτίσει ένα μεταγλωττιστή και να ολοκληρώσει της εργασίες που αναφέρθηκαν πιο πάνω. Υποστηρίζει όλες τις γλώσσες προγραμματισμού που παρέχονται από την πλατφόρμα Visual Studio. Θεωρείται ως ένα σημαντικό εργαλείο το οποίο μπορεί να χρησιμοποιηθεί σε αρκετές εφαρμογές οι οποίες χρειάζονται την επιβεβαίωση ότι το λογισμικό το οποίο θα ελεγχθεί είναι συντακτικά ορθό και μπορεί να εκτελεστεί χωρίς οποιαδήποτε προβλήματα αναφοράς σε βιβλιοθήκες ή άλλα αρχεία με τα οποία συνδέεται. Συνήθως είναι το πρώτο βήμα σε κάθε τεχνική ελέγχου λογισμικών συστημάτων.



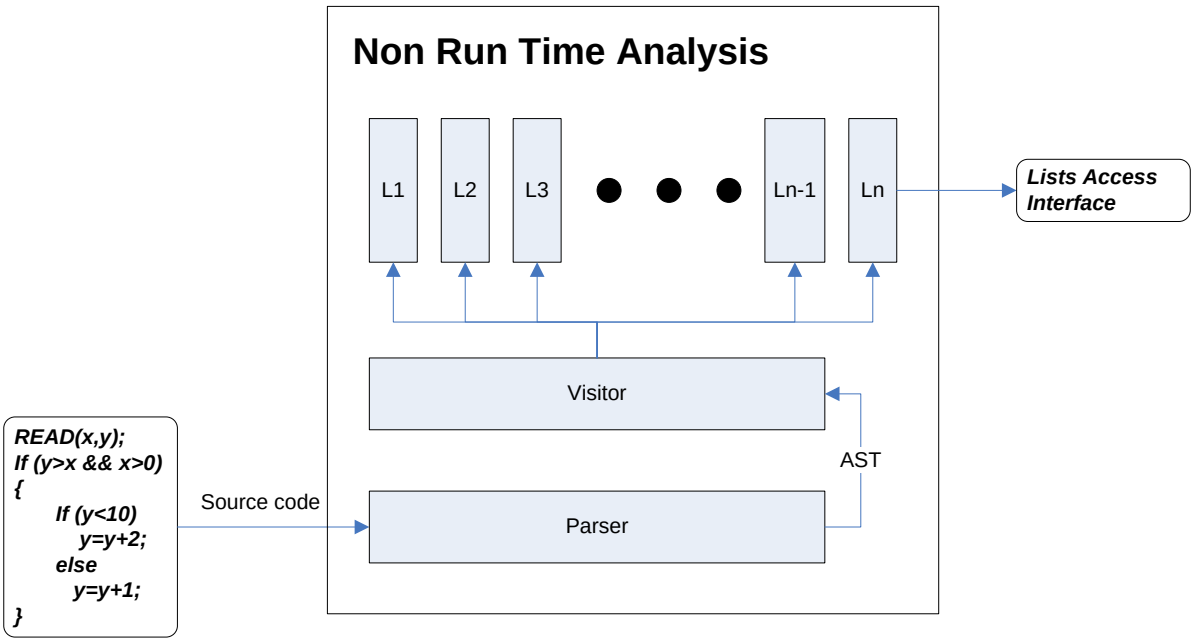
Σχήμα 5.1 – Αρχιτεκτονική κομματιού επικύρωσης πηγαίου κώδικα

5.2.2 Στατική Ανάλυση Πηγαίου Κώδικα

Το δεύτερο συστατικό το οποίο έχει υλοποιηθεί είναι αυτό της στατικής ανάλυσης πηγαίου κώδικα, δηλαδή της ανάλυσης του κώδικα χωρίς αυτό να είναι σε φάση

εκτέλεσης. Με τον όρο ανάλυση εννοούμε την εξαγωγή χρήσιμων πληροφοριών μέσω του πηγαίου κώδικα οι οποίες αφορούν τη δομή του προγράμματος. Δέχεται ως είσοδο τον πηγαίο κώδικα και χρησιμοποιεί την κλάση Abstract Source Tree, η οποία χρησιμοποιήθηκε για την δημιουργία του συστατικού αυτού, μοντελοποιεί την τεχνική αναπαράστασης κώδικα abstract syntax tree και απευθύνεται στην πλατφόρμα Visual Studio 2010 και δημιουργήθηκε από την ομάδα SharpDevelop [11]. Συγκεκριμένα απευθύνεται στις γλώσσες προγραμματισμού C# και Visual Basic. Δεδομένου ενός πηγαίου κώδικα κατά την διάρκεια της μεταγλώττισης του, δημιουργείτε ένα δυαδικό δέντρο, όπου κάθε κόμβος αναπαριστά μία εντολή από τον κώδικα αποθηκεύοντας χρήσιμες πληροφορίες. Έτσι κάποιος μπορεί να διαπεράσει ολόκληρο τον πηγαίο κώδικα μέσω αυτού του δέντρου.

Το συγκεκριμένο συστατικό αποτελείται από δύο υπό-συστήματα, τον parser και τον visitor.



Σχήμα 5.2 – Αρχιτεκτονική στατικού ελέγχου πηγαίου κώδικα

Ο parser είναι υπεύθυνος να αναλύει τον πηγαίο κώδικα λεξικογραφικά και να δημιουργεί ένα Abstract Source Tree. Για την υλοποίηση του parser έχει χρησιμοποιηθεί η βιβλιοθήκη ParserFactory, η οποία υλοποιήθηκε από τους κατασκευαστές του open source IDE για C# και VB.Net projects, SharpDevelop [11].

Συγκεκριμένα η βιβλιοθήκη αυτή πάρθηκε από το SharpDevelop 4.0 [11]. Ο parser δέχεται ως είσοδο τη γλώσσα προγραμματισμού και τον πηγαίο κώδικα.

Ο visitor είναι υπεύθυνος να διαπερνά το Abstract Source Tree και να δίδει τη δυνατότητα στο χρήστη να κάνει τις απαραίτητες προσθήκες έτσι ώστε να αποθηκεύει τις κατάλληλες πληροφορίες τις οποίες θα χρησιμοποιεί ακολούθως. Για την υλοποίηση του visitor έχει χρησιμοποιηθεί η abstract κλάση AbstractAstVisitor, η οποία και αυτή πάρθηκε από το SharpDevelop 4.0 [11]. Η κλάση αυτή δίδει τα βασικά εφόδια για την διαπέραση του Abstract Source Tree και από εκεί και πέρα είναι στη διάθεση του χρήστη να την ενισχύσει για να έχει πρόσβαση σε όλα τα αντικείμενα της γλώσσας που εξετάζει. Στην εργασία αυτή έχει ενισχυθεί έτσι ώστε να διαπερνά όλο το Abstract Source Tree ξεκινώντας από τα υψηλού επιπέδου (high level) χαρακτηριστικά της γλώσσας και να φθάνει μέχρι και τα χαμηλού επιπέδου (low level) χαρακτηριστικά. Επιπλέον κατά την αναδρομική του επίσκεψη σε κάθε κόμβο του Abstract Source Tree αποθηκεύει ένα σύνολο από δεδομένα σε λίστες έτσι ώστε ο χρήστης μετά από μια κλήση του, να έχει όλα τα απαραίτητα δεδομένα για να μπορεί να γνωρίζει τα πάντα για τον πηγαίο κώδικα.

Για να φυλάγονται οι σχετικές πληροφορίες στην ανάλογη λίστα πληροφοριών χρησιμοποιείται μία ενδιάμεση μνήμη (buffer) με τη μορφή στοίβας (stack) μιας και ο visitor διαπερνά ένα δυαδικό δέντρο. Οι λίστες που έχουν ορισθεί είναι οι εξής:

- Λίστα κλάσεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις κλάσεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι το όνομα, ο τύπος (public, static) και ο αριθμός της αρχικής και τελικής γραμμής της κλάσης.

- Λίστα χαρακτηριστικών των κλάσεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τα χαρακτηριστικά των κλάσεων που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση που ανήκουν, το όνομα των μεταβλητών, ο τύπος τους και η τιμή αρχικοποίησής τους (αν υπάρχει).

- Λίστα μεθόδων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις μεθόδους που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι το όνομα, ο τύπος (public, static), η κλάση που ανήκει και ο αριθμός της αρχικής και τελικής γραμμής της μεθόδου.

- Λίστα τοπικών μεταβλητών

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις τοπικές μεταβλητές των μεθόδων που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και μέθοδος που ανήκει, το όνομα της μεταβλητής, ο τύπος της και η τιμή αρχικοποίησης της (αν υπάρχει).

- Λίστα παραμέτρων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις παραμέτρους των μεθόδων που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και μέθοδος που ανήκει, το όνομα της παραμέτρου και ο τύπος της.

- Λίστα προδιαγραφών

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις δηλώσεις προδιαγραφών μέσω των Συμβολαίων Πηγαίου Κώδικα (Code Contracts) που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η δήλωση αυτή, η γραμμή της δήλωσης, ο τύπος της δήλωσης και η συνθήκη της δήλωσης.

- Λίστα αριθμητικών πράξεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις αριθμητικές πράξεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η πράξη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η πράξη.

- Λίστα σχεσιακών πράξεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις σχεσιακές πράξεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η πράξη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η πράξη.

- Λίστα μοναδιαίων πράξεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις μοναδιαίες (unary) πράξεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η πράξη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η πράξη.

- Λίστα μεταβολής ψηφίων πράξεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις πράξεις μετατόπισης (shift) που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η πράξη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η πράξη.

- Λίστα λογικών πράξεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις λογικές πράξεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η πράξη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η πράξη.

- Λίστα συνθηκών

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις συνθήκες που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση και η μέθοδος που ανήκει η συνθήκη, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η συνθήκη.

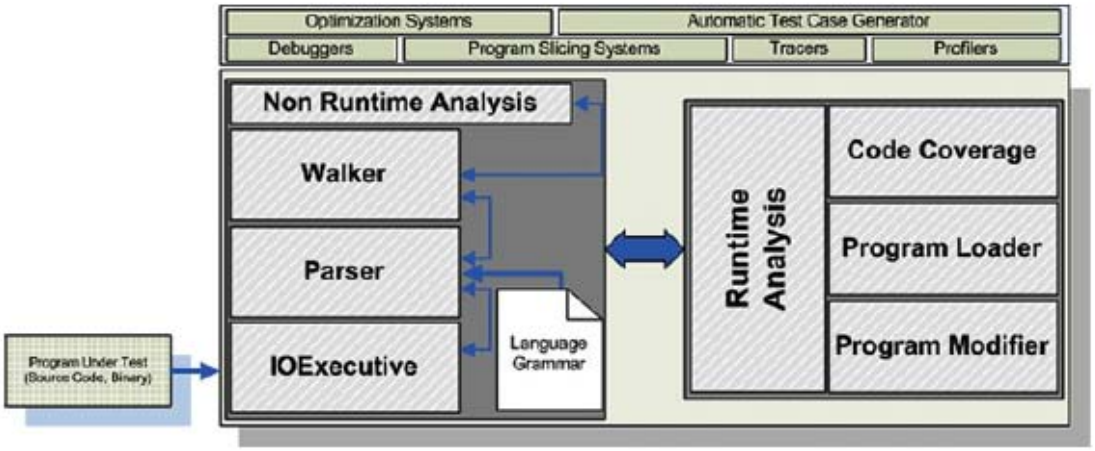
- Λίστα αναθέσεων

  Στη συγκεκριμένη λίστα κρατούνται πληροφορίες σχετικά με τις αναθέσεις που εμφανίζονται στον πηγαίο κώδικα. Μερικές από τις πληροφορίες είναι η κλάση

και η μέθοδος που ανήκει η ανάθεση, η γραμμή που εμφανίζεται στον κώδικα και η έκφραση στην οποία εμφανίζεται η ανάθεση.

Με αυτό τον τρόπο ο χρήστης που θα χρησιμοποιήσει τον visitor, θα έχει την ευχέρεια με ένα κάλεσμά του, να έχει στη διάθεση του όλες αυτές τις πληροφορίες. Επιπλέον η υλοποίηση του visitor είναι αρκετά δομημένη έτσι ώστε ανα πάσα στιγμή να προστίθεται επιπλέον λίστα ή επιπλέον πληροφορίες στις υπάρχουσες λίστες, βάσει των αναγκών του χρήστη.

Και αυτό το συστατικό θεωρείται θεμελιώδες στο χώρο του ελέγχου ενός λογισμικού συστήματος. Η δυνατότητα εφαρμογής στατικής ανάλυσης και η συλλογή τέτοιων πληροφοριών είναι αρκετά βοηθητική για όλο το φάσμα των τεχνικών ελέγχου άσπρου κουτιού. Το συστατικό αυτό μοιάζει κατά πολύ με το στατικό κομμάτι του BPAS το οποίο εμφανίστηκε για πρώτη φορά στην μελέτη [27] ενώ χρησιμοποιήθηκε σε αρκετές προηγούμενες μελέτες. Το BPAS αναφέρεται στη γλώσσα προγραμματισμού Java. Το BPAS έχει χρησιμοποιηθεί σε αυτόματο σύστημα παραγωγής περιπτώσεων δοκιμής (Automatic Test Cases Generation System) [28]. Επίσης χρησιμοποιήθηκε για την ανάπτυξη του γράφου ροής δεδομένων (data flow graph), ο οποίος χτίζεται χρησιμοποιώντας τις πληροφορίες που παρέχονται από το γράφο ροής ελέγχου (control flow graph) του συγκεκριμένου προγράμματος που ελέγχεται [30]. Μία άλλη χρήση ήταν για την δημιουργία ενός συστήματος αυτόματης παραγωγής των βέλτιστων στοιχείων δοκιμής δεδομένου ενός προγράμματος [29]. Το σύστημα αυτό έχει χρησιμοποιηθεί για την αυτόματη δημιουργία γράφου ελέγχου και περιπτώσεις ελέγχου μέσω αλγορίθμων βελτιστοποίησης [31], για τη δημιουργία γράφων εξάρτησης (dependence graph) [32] και για τον αυτοματοποιημένο έλεγχο λογισμικού με βελτιστοποίηση του κριτήριου κάλυψης μονοπατιού [33]. Επίσης έχει χρησιμοποιηθεί και στην υλοποίηση της τεχνικής Symbolic Testing [34] για τον έλεγχο λογισμικών συστημάτων.



Σχήμα 5.3 – Αρχιτεκτονική BPAS [27]

5.3 Αρχιτεκτονική και Ανάλυση της μηχανής παραγωγής μεταλλάξεων

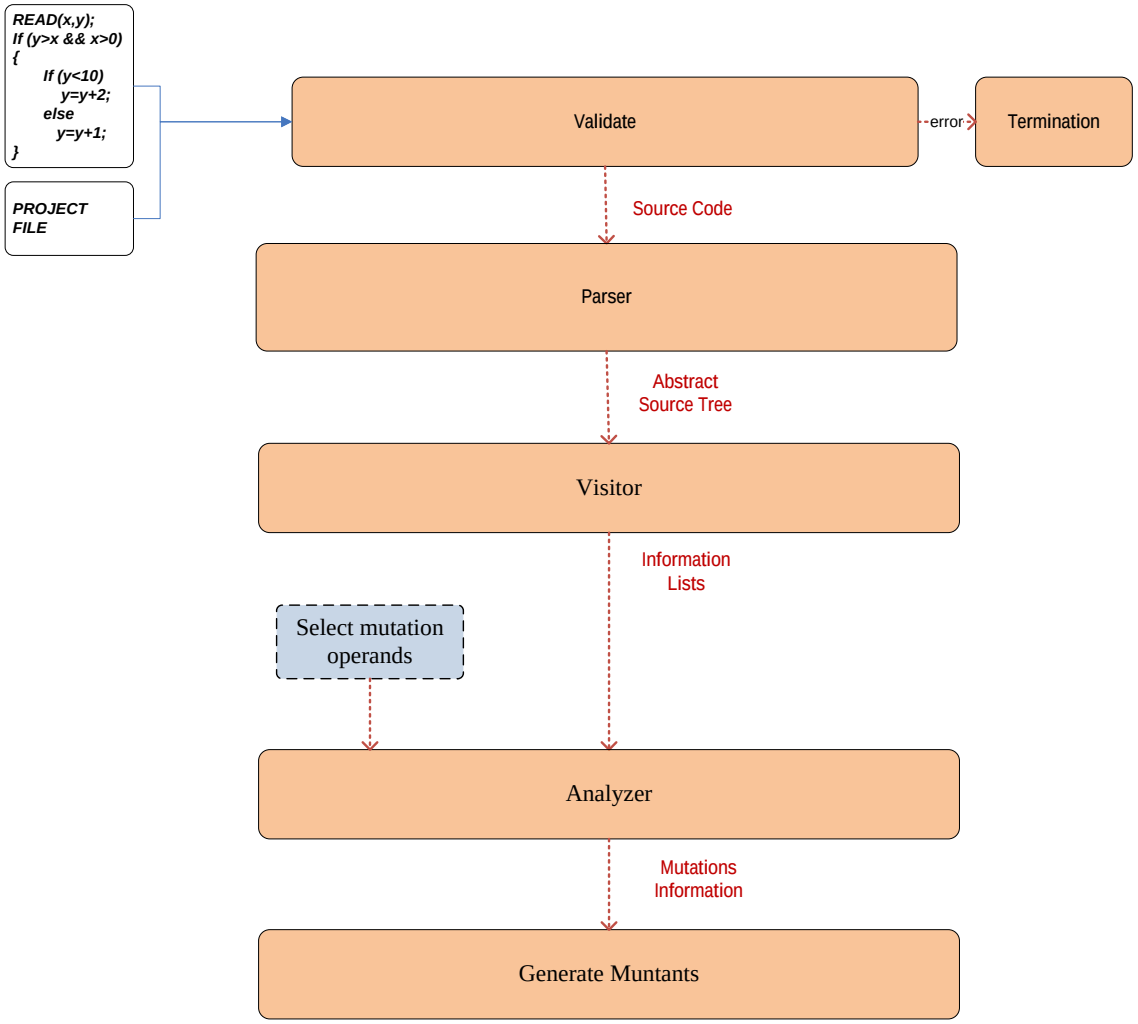
Μετά την υλοποίηση των πιο πάνω συστατικών, έφθασε η ώρα να χρησιμοποιηθούν στην πράξη έτσι ώστε να διαφανεί η χρησιμότητά τους καθώς και ο τρόπος ενσωμάτωσής τους σε μια νέα εφαρμογή. Η απουσία μιας μηχανής παραγωγής μεταλλάξεων για την πλατφόρμα Visual Studio 2010 και το .NET framework 4.0, καθώς και οι προοπτικές χρήσης μιας τέτοιας μηχανής, κίνησε το ενδιαφέρον για υλοποίησή της. Η μηχανή που υλοποιήθηκε ασχολείται με μεταλλάξεις σε επίπεδο μεθόδων. Η αρχιτεκτονική της όμως είναι δομημένη με τέτοιο τρόπο ώστε να μπορεί κάποιος να ενσωματώσει και επιπέδου κλάσης τελεστές μετάλλαξης (όλες οι γλώσσες προγραμματισμού που παρέχει το Visual Studio είναι Object Oriented) αλλά και να επεκτείνει αν κρίνει σκόπιμο τους ήδη υπάρχοντες τελεστές μετάλλαξης. Στην τρέχουσα έκδοση η μηχανή επεξεργάζεται μόνο προγράμματα γραμμένα στη γλώσσα προγραμματισμού C#, ενώ με κάποιες αλλαγές θα μπορεί να υποστηρίξει και άλλες γλώσσες.

5.3.1 Αρχιτεκτονική της μηχανής παραγωγής μεταλλάξεων

Η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα αποτελείται από τρία βασικά μέρη, το κομμάτι επικύρωσης της ορθότητας του λογισμικού υπό μετάλλαξη, το κομμάτι στατικής ανάλυσης του λογισμικού υπό μετάλλαξη και το κομμάτι παραγωγής μεταλλάξεων. Όπως θα παρατηρήσετε τα δύο πρώτα μέρη χρησιμοποιούν τα ήδη υλοποιημένα συστατικά τα οποία παρουσιάστηκαν στο προηγούμενο υποκεφάλαιο.

Το τρίτο κομμάτι, αυτό της παραγωγής μεταλλάξεων, ασχολείται με την ανάλυση των πληροφοριών που δημιούργησε η στατική ανάλυση του πηγαίου κώδικα, με σκοπό την πλήρη γνώση της δομής και του περιεχομένου του. Οι πληροφορίες βρίσκονται στις ανάλογες λίστες και διατηρούν μία δομή έτσι ώστε να μπορούν να αναλυθούν από το κομμάτι παραγωγής μεταλλάξεων. Η ανάλυσή τους γίνεται με μία ενδιάμεση μνήμη (buffer) υπό μορφή στοίβας (stack) γιατί με αυτό τον τρόπο αποθηκεύονται από τον visitor. Μετά την ανάλυση των πληροφοριών, το κομμάτι αυτό έχει τη δυνατότητα παραγωγής μεταλλαγμένων προγραμμάτων βάσει ενός συνόλου από προκαθορισμένους τελεστές μετάλλαξης. Οι τελεστές μετάλλαξης θα αναλυθούν στο επόμενο υποκεφάλαιο.

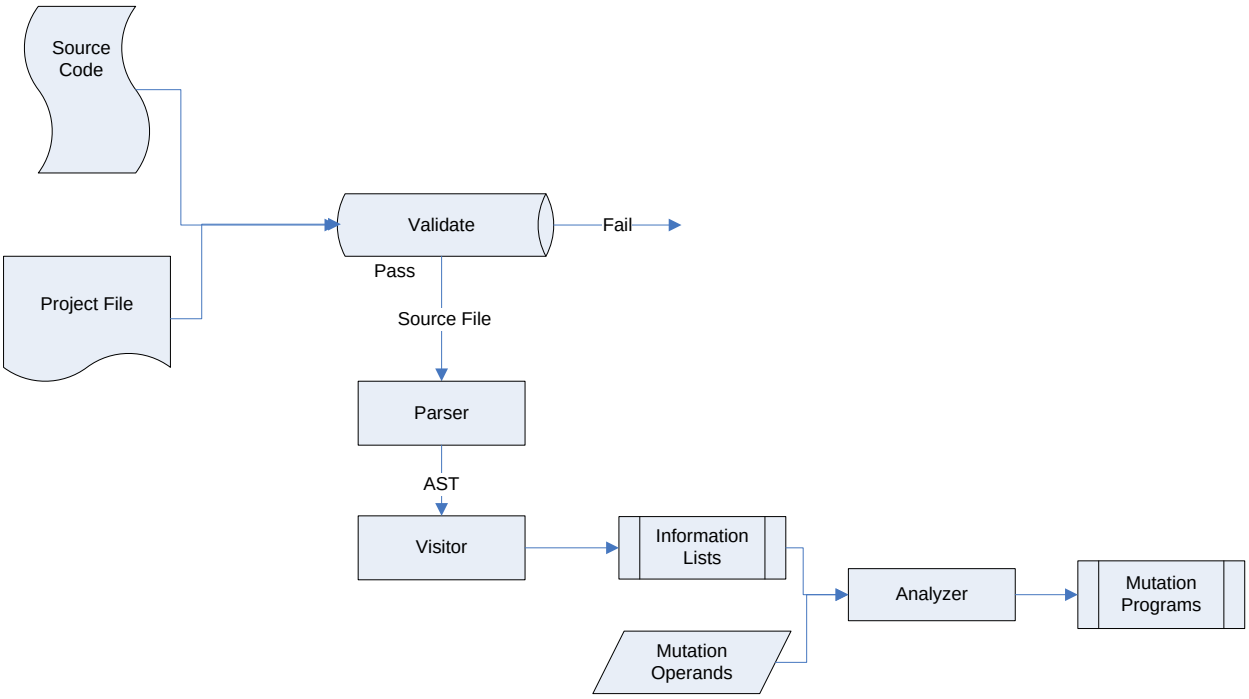
Ο αλγόριθμος με τον οποίο δουλεύει η μηχανή παραγωγής μεταλλάξεων είναι ο εξής:



Σχήμα 5.4 – Σχήμα του υλοποιημένου Mutation Engine

Για περαιτέρω κατανόηση του αλγορίθμου, ακολουθεί εκτενής επεξήγησή του.

Το σύστημα παίρνει σαν είσοδο τον πηγαίο κώδικα μαζί με το project file της εφαρμογής στην οποία ανήκει το λογισμικό υπό μετάλλαξη. Στη συνέχεια, μεταγλωττίζει τον πηγαίο κώδικα μέσω του κομματιού επικύρωσης, βρίσκοντας όλες τις απαραίτητες πληροφορίες από το project file, με σκοπό την επιβεβαίωση της ορθότητάς του. Σε περίπτωση παρουσίασης σφάλματος στον πηγαίο κώδικα, το σύστημα δεν προχωρά στο επόμενο βήμα βάσει του αλγορίθμου του και ενημερώνει μέσω ενός pop-up παράθυρου τους λόγους τερματισμού της λειτουργίας του. Σε περίπτωση ύπαρξης προειδοποιήσεων, τα γνωστά σε όλους warnings, το σύστημα εμφανίζει το σχετικό pop-up για να ενημερώσει τον χρήστη και συνεχίζει κανονικά την λειτουργία του. Όταν δεν υπάρχουν σφάλματα, το σύστημα εφαρμόζει στατική ανάλυση του πηγαίου κώδικα. Δηλαδή, μέσω του κομματιού στατικής ανάλυσης το οποίο λαμβάνει σαν είσοδο τον πηγαίο κώδικα, αναλύεται λεξικογραφικά ο κώδικας αυτός και δημιουργείται ένα Abstract Source Tree το οποίο αναπαριστά υπό τη μορφή δυαδικού δέντρου όλες τις εντολές που υπάρχουν στον πηγαίο κώδικα. Το Abstract Source Tree που δημιουργείται από τον parser χρησιμοποιείται από τον visitor ο οποίος το διαπερνά αποθηκεύοντας χρήσιμες πληροφορίες στις λίστες δεδομένων που υπάρχουν. Ακολούθως το τρίτο κομμάτι παίρνει σαν είσοδο τις λίστες δεδομένων καθώς και ένα σύνολο τελεστών μετάλλαξης και παράγει τα μεταλλαγμένα προγράμματα.



Σχήμα 5.5 – Αρχιτεκτονική του υλοποιημένου Mutation Engine

5.3.2 Τελεστές μετάλλαξης πηγαίου κώδικα

Το εργαλείο παραγωγής μεταλλαγμένων λογισμικών συστημάτων παρέχει ένα σύνολο από προκαθορισμένους τελεστές μετάλλαξης. Οι τελεστές αυτοί είναι υπεύθυνοι για την μετάλλαξη του πηγαίου κώδικα βάσει ενός κανόνα που αντιπροσωπεύουν, χωρίς να παραβιάζουν την ορθότητα του μεταλλαγμένου προγράμματος ως προς τη γραμματική της εκάστοτε γλώσσας προγραμματισμού. Στο υποκεφάλαιο αυτό θα παρουσιαστούν αυτοί οι τελεστές που υπάρχουν στη γλώσσα προγραμματισμού C# καθώς και προκαθορισμένοι τελεστές μετάλλαξης του εργαλείου.

Η γλώσσα προγραμματισμού C# παρέχει τους πιο κάτω τελεστές:

Κατηγορία	Τελεστές
Unary	+ - ! ~
Shortcut	++x --x x++ x--
Multiplicative	* / %
Additive	+ -
Shift	<< >>
Relational	< > <= >=
Relational – Equality	== !=
Logical AND	&
Logical XOR	^
Logical OR	|
Conditional AND	&&
Conditional OR	||
Assignment	= *= /= %= += -= <<= >>= &= ^= |=

Πίνακας 5.1 – τελεστές που προσφέρει η πλατφόρμα Visual Studio

Οι προκαθορισμένοι τελεστές μετάλλαξης που παρέχει το υλοποιημένο mutation engine είναι οι εξής:

- Αριθμητικές Πράξεις
    - **AOR_{BA} – arithmetic operations replacement (binary, assignment)**

      Ο τελεστής μετάλλαξης **AOR_{BA}** αναφέρεται στην αντικατάσταση των binary αριθμητικών πράξεων +, -, *, / και %. Επίσης αντικαθιστά τα προηγούμενα σύμβολα στις αναθέσεις +=, -=, *=, /= και %=.

- **AOR_S – arithmetic operations replacement (shortcut)**

Ο τελεστής μετάλλαξης **AOR_S** αναφέρεται στην αντικατάσταση των shortcut αριθμητικών πράξεων ++ και --. Η αντικατάσταση γίνεται για όλες τις θέσεις που μπορούν να εμφανιστούν οι shortcut αριθμητικές πράξεις, δηλαδή πριν και μετά από την αριθμητικού τύπου μεταβλητή.

- **AOI_S – arithmetic operations insertion (shortcut)**

Ο τελεστής μετάλλαξης **AOI_S** αναφέρεται στην εισαγωγή των shortcut αριθμητικών πράξεων ++ και -- πριν από κάθε αριθμητικού τύπου μεταβλητές. Δεν εισάγονται πίσω από την μεταβλητή γιατί η αλλαγή δεν θα έχει αξία για εκείνη την γραμμή αλλά για την επόμενη, η οποία μπορεί να καλυφθεί με την επόμενη μετάλλαξη όπου θα προστεθεί στην επόμενη παρουσίαση της μεταβλητής.

- **AOI_U – arithmetic operations insertion (unary)**

Ο τελεστής μετάλλαξης **AOI_U** αναφέρεται στην εισαγωγή του τελεστή – μπροστά από κάθε αριθμητικού τύπου μεταβλητές. Εάν υπάρχει ήδη ο τελεστής αυτός δεν θα προστεθεί. Επίσης σε unsigned τύπους μεταβλητές δεν εισάγεται ο τελεστής -.

- **AOI_A – arithmetic operations insertion (assignment)**

Ο τελεστής μετάλλαξης **AOI_A** αναφέρεται στην εισαγωγή των binary αριθμητικών πράξεων +, -, *, / και % σε αναθέσεις. Δηλαδή όπου υπάρχει αριθμητική ανάθεση π.χ. a = b ή a = b + 2, τότε εάν η μεταβλητή στο αριστερό μέρος της ανάθεσης είναι αρχικοποιημένη, ο τελεστής εισάγει τα πιο πάνω σύμβολα.

- **AOD_S – arithmetic operations deletion (shortcut)**

Ο τελεστής μετάλλαξης **AOD_S** αναφέρεται στη διαγραφή των shortcut αριθμητικών πράξεων ++ και -- οι οποίες βρίσκονται πριν ή μετά από κάθε αριθμητική πράξη.

- **AOD_U – arithmetic operations deletion (unary)**

Ο τελεστής μετάλλαξης **AOD_U** αναφέρεται στη διαγραφή του συμβόλου – πριν από κάθε αριθμητικού τύπου μεταβλητές.

- **AOD_A – arithmetic operations deletion (assignment)**

  Ο τελεστής μετάλλαξης **AOD_A** αναφέρεται στη διαγραφή των binary αριθμητικών πράξεων +, -, *, / ,% στις αναθέσεις +=, -=, *=, /= και %=.

- Σχεσιακές Πράξεις
  - **ROR – relational operations replacement**

    Ο τελεστής μετάλλαξης **ROR** αναφέρεται στην αντικατάσταση των σχεσιακών πράξεων >,<,< =,> =, = =, και ! =.

- Πράξεις Συνθηκών
  - **COR – conditional operations replacement**

    Ο τελεστής μετάλλαξης **COR** αναφέρεται στην αντικατάσταση των πράξεων συνθήκης && και ||.

  - **COI – conditional operations insertion**

    Ο τελεστής μετάλλαξης **COI** αναφέρεται στην εισαγωγή του συμβόλου ! στις πράξεις συνθηκών καθώς και στις μεταβλητές τύπου boolean.

  - **COD – conditional operations deletion**

    Ο τελεστής μετάλλαξης **COD** αναφέρεται στη διαγραφή του συμβόλου ! από τις πράξεις συνθηκών καθώς και τις μεταβλητές τύπου boolean.

- Λογικές Πράξεις
  - **LOR – logical operations replacement**

    Ο τελεστής μετάλλαξης **LOR** αναφέρεται στην αντικατάσταση των λογικών πράξεων &, | και ^.

  - **LOI – logical operations insertion**

    Ο τελεστής μετάλλαξης **LOI** αναφέρεται στην εισαγωγή του λογικού τελεστή ~, πριν από μεταβλητές τύπου Integer και Long.

- **LOI_A – logical operations insertion (assignment)**

  Ο τελεστής μετάλλαξης **LOI_A** αναφέρεται στην εισαγωγή των λογικών πράξεων &, | και ^ στις λογικές αναθέσεις. Πριν την εισαγωγή των συμβόλων αυτών, γίνεται ο έλεγχος εάν είναι αρχικοποιημένη η μεταβλητή του αριστερού μέλους της ανάθεσης, αν όχι δεν γίνονται οι εισαγωγές των ψηφίων.

- **LOD – logical operations deletion**

  Ο τελεστής μετάλλαξης **LOD** αναφέρεται στη διαγραφή του λογικού τελεστή ~, πριν από μεταβλητές τύπου Integer και Long.

- **LOD_A – logical operations deletion (assignment)**

  Ο τελεστής μετάλλαξης **LOD_A** αναφέρεται στη διαγραφή των λογικών πράξεων &, | και ^ από τις λογικές αναθέσεις.

- Πράξεις Μεταποιήσεων
  - **SOR – shift operations replacement**

    Ο τελεστής μετάλλαξης **SOR** αναφέρεται στην αντικατάσταση των πράξεων μετατόπισης ψηφίων >> και <<.

  - **SOI_A – shift operations insertion (assignment)**

    Ο τελεστής μετάλλαξης **SOI_A** αναφέρεται στην εισαγωγή των πράξεων μετατόπισης ψηφίων >> και <<. Πριν την εισαγωγή των συμβόλων αυτών, γίνεται ο έλεγχος εάν είναι αρχικοποιημένη η μεταβλητή του αριστερού μέλους της ανάθεσης, αν όχι δεν γίνονται οι εισαγωγές των ψηφίων.

  - **SOD_A – shift operations deletion (assignment)**

    Ο τελεστής μετάλλαξης **SOD_A** αναφέρεται στη διαγραφή των πράξεων μετατόπισης ψηφίων >> και <<

- Πράξεις Εναλλαγής Μεταβλητών
    - **PR – parameter replacement**

      Ο τελεστής **PR** αναφέρεται στην εναλλαγή δύο παραμέτρων ιδίου τύπου στη γραμμή όπου δηλώνεται μία μεταβλητή. Εάν υπάρχουν περισσότερες από δύο μεταβλητές, τότε εκτελείται η πιο πάνω διαδικασία ανά ζεύγη μέχρι να καλυφθούν όλες οι περιπτώσεις. Επίσης εναλλαγή υπάρχει και στην μέθοδο μεταξύ του ζεύγους των μεταβλητών. Κάθε εναλλαγή αποτιμάται ως μία μετάλλαξη.

    - **LVR – local variable replacement**

      Ο τελεστής **LVR** αναφέρεται στην εναλλαγή δύο τοπικών μεταβλητών ιδίου τύπου. Εάν υπάρχουν περισσότερες από δύο μεταβλητές, τότε εκτελείται η πιο πάνω διαδικασία ανά ζεύγη μέχρι να καλυφθούν όλες οι περιπτώσεις. Κάθε εναλλαγή αποτιμάται ως μία μετάλλαξη.

Όλες οι μεταλλάξεις αποθηκεύονται στο μονοπάτι C:/Mutations/ σε φακέλους με τους οποίους αναγνωρίζεται ο τελεστής που προκάλεσε το κάθε μεταλλαγμένο πρόγραμμα. Επίσης η τελευταία γραμμή στον κώδικα σημειώνει σε μορφή σχολίων την αλλαγή που έχει υποστεί ο αρχικός πηγαίος κώδικας.

5.4 Καινοτόμος Ιδέα για βελτιστοποίηση της μηχανής παραγωγής μεταλλάξεων

Είναι γεγονός πως η μηχανή παραγωγής μεταλλάξεων παράγει ένα μεγάλο αριθμό από μεταλλαγμένα προγράμματα, κάνοντας την περαιτέρω επεξεργασία τους μια χρονοβόρα διαδικασία η οποία χρειάζεται μεγάλη υπολογιστική ισχύ. Σε κάποιες περιπτώσεις, ο μεγάλος αριθμός μεταλλαγμένων προγραμμάτων είναι επιθυμητός και σε κάποιες άλλες όχι. Για παράδειγμα, για να αποτιμηθεί ο βαθμός καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής, όσα περισσότερα μεταλλαγμένα προγράμματα έχουμε, τόσο πιο ενισχυμένη θα είναι η διαδικασία από την οποία αναμένουμε αρκετά καλά αποτελέσματα. Από την άλλη όμως, εάν η μηχανή παραγωγής μεταλλάξεων χρησιμοποιείται ως ένα βοηθητικό εργαλείο για την ανεύρεση σφαλμάτων, πρέπει το σύνολο των μεταλλαγμένων προγραμμάτων να περιορίζεται σε αυτά που πιθανόν να περιέχουν το σφάλμα και να απαλείφονται τα υπόλοιπα.

Μέσα από την μελέτη της χρησιμότητας της μηχανής παραγωγής μεταλλάξεων καθώς και των δυνατοτήτων της πλατφόρμας Visual Studio 2010, προσφέρεται η δυνατότητα ενσωμάτωσης «εξυπνάδας» στη μηχανή. Σε προηγούμενο κεφάλαιο έγινε αναφορά για την δυνατότητα ενσωμάτωσης των προδιαγραφών ενός λογισμικού συστήματος στον ίδιο τον πηγαίο κώδικα με δυνατότητα αλληλεπίδρασης με τον υπόλοιπο κώδικα. Μπορούμε να ενισχύσουμε την μηχανή παραγωγής μεταλλάξεων με επιπλέον ελέγχους έτσι ώστε κομμάτια κώδικα τα οποία αποτελούν μέρος των προδιαγραφών να μην μεταλλάζονται. Επίσης επιπλέον έλεγχοι γίνονται έτσι ώστε να μην υπάρχουν μεταλλαγμένα προγράμματα τα οποία να έχουν συντακτικά σφάλματα, όπως για παράδειγμα ανάθεση σε μη αρχικοποιημένες μεταβλητές.

Τα συμβόλαια πηγαίου κώδικα (Code Contracts), δίδουν την δυνατότητα στο χρήστη να καθορίζει τις προαπαιτούμενες (pre-conditions), απαιτούμενες (post-conditions) και σταθερές (invariant) συνθήκες. Επίσης το Visual Studio μέσω κάποιων ρυθμίσεων μπορεί να εκτελεί στατική και δυναμική ανάλυση έτσι ώστε να ενημερώνει τον χρήστη πότε παραβιάζονται οι συνθήκες που έχει δηλώσει και να σταματά την ομαλή εκτέλεση του λογισμικού συστήματος. Στην πιο κάτω εικόνα παρουσιάζεται ένα παράδειγμα ενσωμάτωσης των προδιαγραφών στον πηγαίο κώδικα.

```
1 using System;
2 using System.Diagnostics.Contracts;
3
4 namespace ContractExample1 {
5
6   class Rational {
7
8     int numerator;
9     int denominator;
10
11    public Rational(int numerator, int denominator)
12    {
13      Contract.Requires( denominator != 0 );
14
15      this.numerator = numerator;
16      this.denominator = denominator;
17    }
18
19    public int Denominator {
20      get {
21        Contract.Ensures( Contract.Result<int>() != 0 );
22
23        return this.denominator;
24      }
25    }
26
27    [ContractInvariantMethod]
28    void ObjectInvariant () {
29      Contract.Invariant ( this.denominator != 0 );
30    }
31  }
32 }
```

Εικόνα 5.1 – Πηγαίος κώδικας εμπλουτισμένος με εντολές ενσωμάτωσης προδιαγραφών

Με την εντολή Contract.Requires ο χρήστης δηλώνει τα προαπαιτούμενα (pre-conditions). Στο συγκεκριμένο παράδειγμα παρατηρούμε ότι η παράμετρος denominator δεν μπορεί ποτέ να πάρει την τιμή μηδέν.
Με την εντολή Contract.Ensures ο χρήστης δηλώνει τα απαιτούμενα (post-conditions). Στο συγκεκριμένο παράδειγμα παρατηρούμε ότι η μέθοδος Denominator δεν μπορεί να επιστρέψει τιμή ίση με μηδέν.
Με την εντολή Contract.Invariant ο χρήστης δηλώνει τις σταθερές (invariants) οι οποίες διέπουν την συγκεκριμένη κλάση. Στο συγκεκριμένο παράδειγμα παρατηρούμε ότι το χαρακτηριστικό της κλάσης Denominator, με το όνομα denominator, δεν μπορεί να πάρει την τιμή μηδέν.

Εάν θεωρηθεί ότι ο χρήστης τοποθετεί τις ορθές προδιαγραφές στον πηγαίο κώδικα, τότε η μετάλλαξη του συγκεκριμένου κώδικα μπορεί πλέον να μην είναι μια μηχανική

διαδικασία, αλλά να καθοδηγείται μέσω των προδιαγραφών που έχουν δηλωθεί. Με λίγα λόγια, η κάθε υποψήφια αλλαγή βάσει των τελεστών μετάλλαξης, μπορεί να ελεγχθεί εάν παραβιάζει τις προδιαγραφές μέσω μιας στατικής ανάλυσης και θα αποφασίζεται εάν υπάρχει η δυνατότητα εφαρμογής της αλλαγής ή όχι. Οι προδιαγραφές δεν περιγράφουν μόνο τον υπό ανάπτυξη/έλεγχο κώδικα αλλά και τις διαπροσωπείες μεταξύ άλλων μεθόδων από την ίδια ή διαφορετική κλάση. Ο τρόπος ελέγχου απαιτεί στατική ανάλυση για τον έλεγχο των αλλαγών μιας και δεν θα υπάρχουν περιθώρια εκτέλεσης του μεταλλαγμένου προγράμματος, γιατί μετά το κόστος θα ήταν αρκετά ψηλό και δεν θα παρουσιαζόταν οποιαδήποτε βελτίωση.

Παράδειγμα εφαρμογής ιδέας:

```
public class Test
{
        private int Foo(int a, int b)
        {
                Contract.Requires(a > b);
                Contract.Requires(b > 0);
                Contract.Ensures( Contract.Result <int>() > 0);
                ….
                ….
                return (a / b);
        }
        private void Goo( )
        {
                int x, y;
                ….
                x = y + 10;
                int result = Foo (x , y)
        }
}
```

Στο πιο πάνω παράδειγμα παρουσιάζεται μία κλάση, η Test, η οποία αποτελείται από δύο μεθόδους, την Foo και την Goo, και χρησιμοποιεί τα συμβόλαια πηγαίου κώδικα

(Code Contracts) για την αποτύπωση των προδιαγραφών στον πηγαίο κώδικα. Από πλευράς προδιαγραφών παρατηρείται η ύπαρξη δύο προαπαιτούμενων (pre-condtions) στην κλάση Foo, το a > b και το b > 0 και ένα απαιτούμενο (post-condition) το οποίο επιβεβαιώνει ότι το αποτέλεσμα που θα επιστρέψει η μέθοδος Foo θα είναι θετικό και μη μηδενικό.

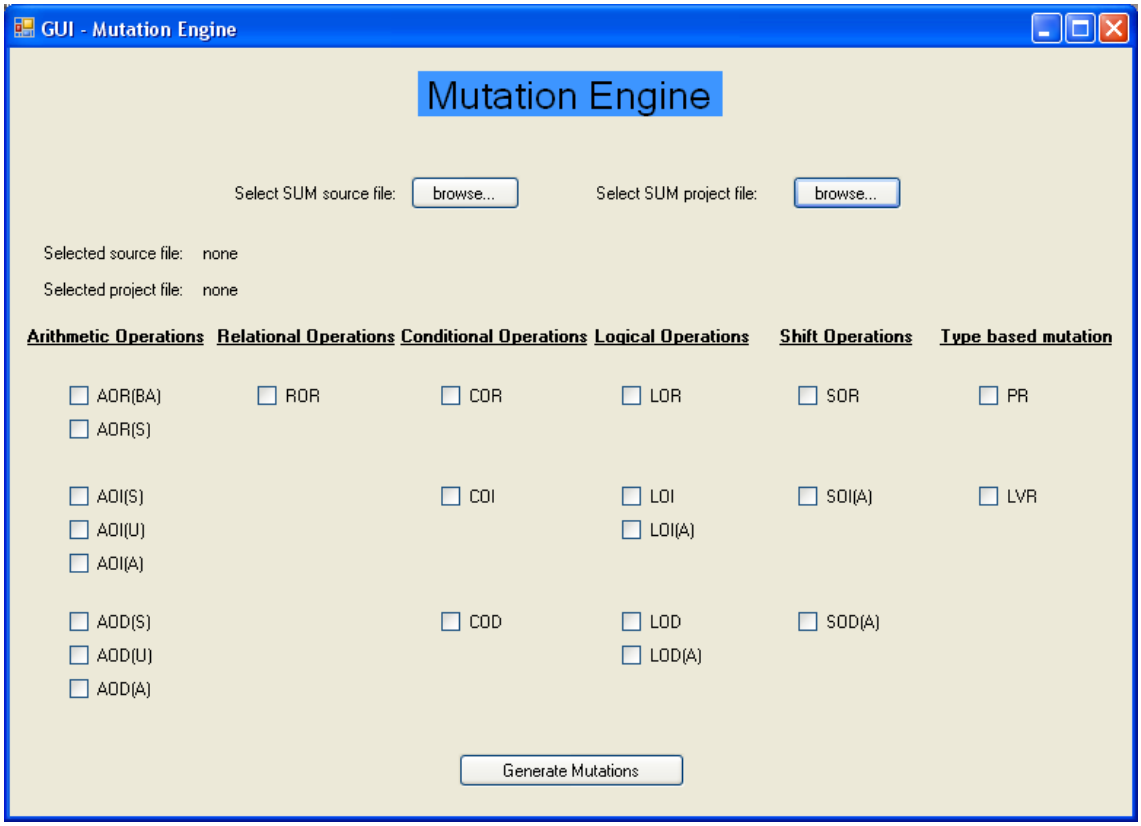
Κάποιος θα αναρωτηθεί πώς μπορούν αυτές οι τρεις πληροφορίες να κατευθύνουν την διαδικασία μετάλλαξης. Αρχικά, στην κλάση Goo η γραμμή x = y + 10; επηρεάζει την κλήση της μεθόδου όπου χαρακτηρίζεται από δύο προαπαιτούμενα (pre-conditions), δηλαδή στην αρχή της μεθόδου, που πρέπει να ισχύουν για να ολοκληρωθεί η ομαλή λειτουργία του κώδικα. Επομένως, μιας και πρέπει να ισχύει η συνθήκη a > b, είναι προφανές πως πρέπει να ισχύει x > y. Ο τελεστής αντικατάστασης αριθμητικών πράξεων θα αναγνωρίσει την εξίσωση x = y + 10; και θα προτείνει την αντικατάσταση του συμβόλου + με τα σύμβολα -, *, /, %. Βάσει των προδιαγραφών όμως, η ανταλλαγή του συμβόλου + με τα σύμβολα - , / και % είναι λανθασμένη γιατί δεν θα ισχύει πλέον το x > y και κατ'επέκταση το προαπαιτούμενο (pre-condition) a > b της Foo. Επίσης δεν μπορεί να εισαχθεί το πρόσημο – μπροστά από την μεταβλητή y. Το ίδιο όμως ισχύει και για το δεύτερο προαπαιτούμενο (pre-condition) b > 0, δηλαδή οποιαδήποτε μετάλλαξη η οποία το μηδενίζει ή το κάνει αρνητικό δεν θα γίνεται αποδεκτή από το συμβόλαιο πηγαίου κώδικα (Code Contract). Ακολούθως, αν λάβουμε υπόψιν το αναμενόμενο (post-condition) το οποίο εγγυάται ότι η τιμή του αποτελέσματος της Foo πρέπει να είναι θετική, τότε προφανώς δεν μπορεί να εισαχθεί το πρόσημο – μπροστά από τις μεταβλητές a και b. Επίσης, η ανταλλαγή των παραμέτρων θα οδηγούσε σε καταπάτηση του αναμενόμενου (post-condition).

Το πιο πάνω είναι ένα απλό παράδειγμα ενσωμάτωσης των συμβόλαιων πηγαίου κώδικα (Code Contracts) σε πηγαίο κώδικα, καθώς και η επιρροή που ασκούν στη διαδικασία παραγωγής μεταλλάξεων πηγαίου κώδικα. Με μία απλή ανάλυση ενός σχετικά μικρού μεγέθους κώδικα, αποτρέψαμε τη διαδικασία από το να υλοποιήσει ένα σημαντικό αριθμό από «αχρείαστα» μεταλλαγμένα προγράμματα. Με γνώμονα το πιο πάνω συμπέρασμα παρατηρούμε ότι η ιδέα αυτή μπορεί να εφαρμοστεί σε μία μηχανή παραγωγής μεταλλάξεων, προσδίδοντάς της «εξυπνάδα» και αποτρέποντας την παραγωγή μεταλλαγμένων προγραμμάτων τα οποία δεν ικανοποιούν τις προδιαγραφές

του συστήματος. Αυτή η ιδέα μπορεί να χρησιμοποιηθεί σε περιπτώσεις όπου θέλουμε να δημιουργήσουμε ένα σύνολο από μεταλλαγμένα προγράμματα τα οποία ακολουθούν τις προδιαγραφές με αποτέλεσμα η λειτουργία τους να είναι αρκετά κοντά στην αρχική. Όπως αναφέρθηκε και προηγουμένως μία χρήση η οποία αναμένεται να βοηθηθεί από την ιδέα αυτή, είναι ο εντοπισμός σφάλματος σε πηγαίο κώδικα.

5.5 Παρουσίαση του Mutation Engine

Σε αυτό το υποκεφάλαιο θα παρουσιαστεί η διαπροσωπεία του εργαλείου που έχει υλοποιηθεί.



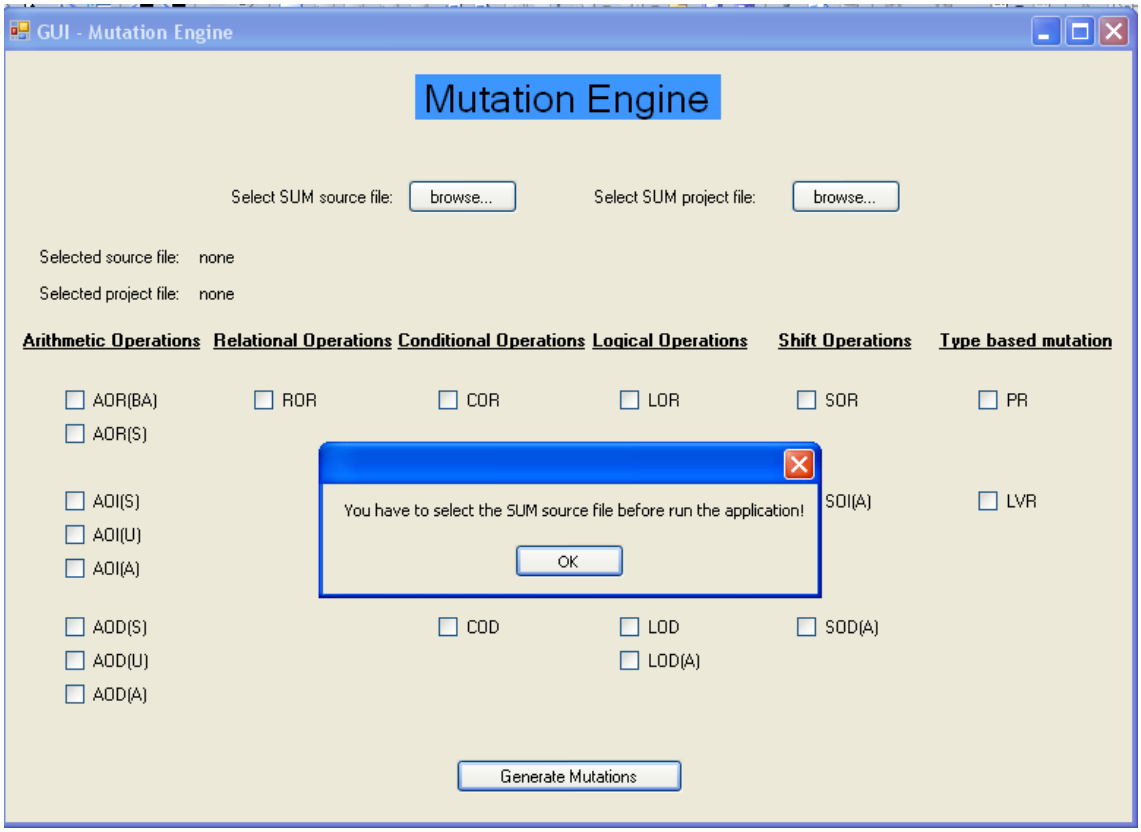
Εικόνα 5.2 – GUI του υλοποιημένου mutation engine

Ο χρήστης καλείται να δώσει σαν είσοδο στο εργαλείο τον πηγαίο κώδικα ο οποίος είναι υπό μετάλλαξη καθώς και το project αρχείο όπου ανήκει ο πηγαίος κώδικας. Μέσω των δύο κουμπιών με την αναγραφή «browse…» μπορεί να αναζητήσει τα μονοπάτια όπου βρίσκονται τα δύο αρχεία. Όταν επιλέξει τα δύο αρχεία το εργαλείο θα τα εμφανίσει έτσι ώστε ο χρήστης να επιβεβαιώσει την επιλογή του πριν προχωρήσει στο επόμενο βήμα. Ακολούθως ο χρήστης πρέπει να διαλέξει ένα ή περισσότερους

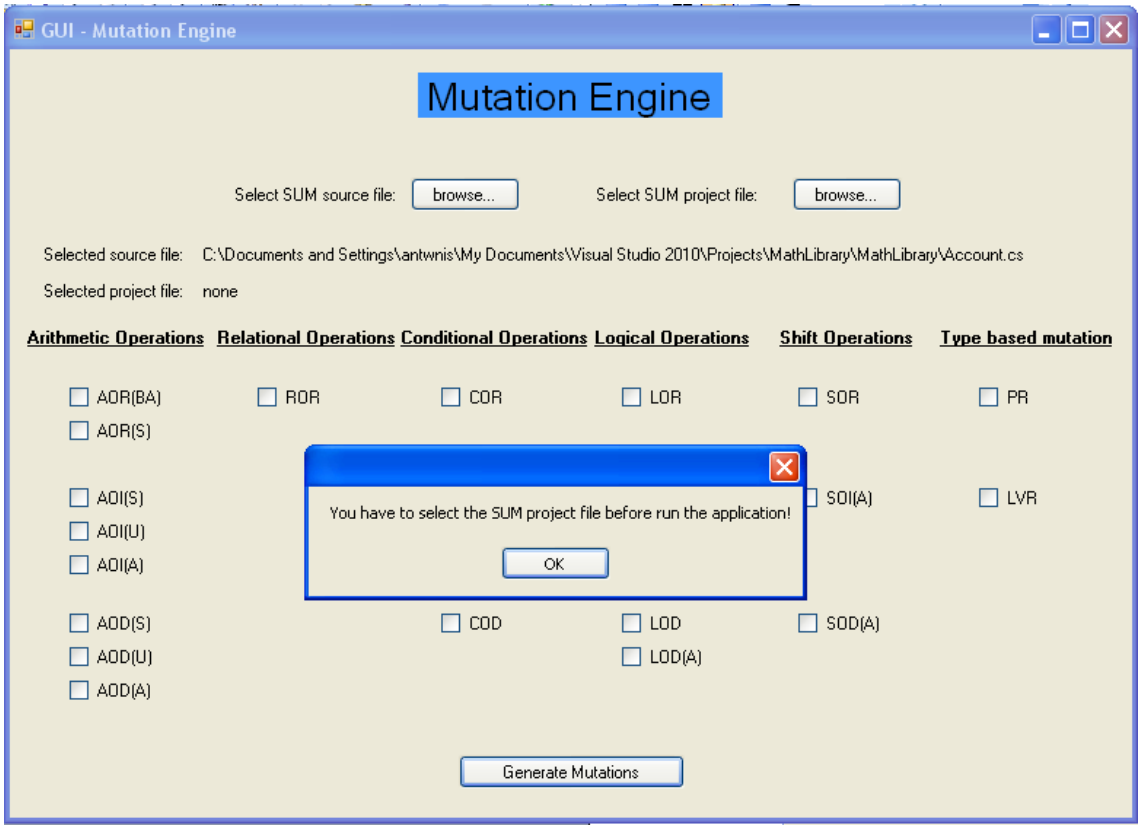
τελεστές μετάλλαξης μέσω των κουτιών επιλογής (checkboxes) που υπάρχουν στην οθόνη. Ο κάθε τελεστής δημιουργεί ξεχωριστά μεταλλαγμένα προγράμματα ενσωματώνοντας μόνο ένα ατομικό σφάλμα κάθε φορά. Οι τελεστές μετάλλαξης είναι ομαδοποιημένοι τόσο κάθετα όσο και οριζόντια για να διευκολύνουν την εύρεσή τους από τον χρήστη. Τέλος, για να τεθεί σε λειτουργία η μηχανή παραγωγής μεταλλάξεων ο χρήστης πρέπει να πατήσει το κουμπί με την αναγραφή «Generate Mutations».

Το εργαλείο ενημερώνει τον χρήστη για οποιαδήποτε ανώμαλη λειτουργία, όπως για παράδειγμα την απουσία ενός από των δυο εισόδων ή και την απουσία επιλογής τουλάχιστον ενός τελεστή μετάλλαξης. Επίσης εάν το πρόγραμμα περιέχει κάποιο συντακτικό λάθος ή το εργαλείο δεν είναι σε θέση να εντοπίσει τις βιβλιοθήκες που χρησιμοποιεί τότε η μετάλλαξη δεν θα ολοκληρωθεί και θα ενημερωθεί ο χρήστης για τα αίτια της μη ολοκλήρωσης της κανονικής λειτουργίας του εργαλείου. Στο τέλος ης λειτουργίας του, το εργαλείο ενημερώνει τον χρήστη για την επιτυχή εκτέλεσή του, καθώς και για την τοποθεσία όπου μπορεί να μελετήσει τα αποτελέσματα της εκτέλεσης.

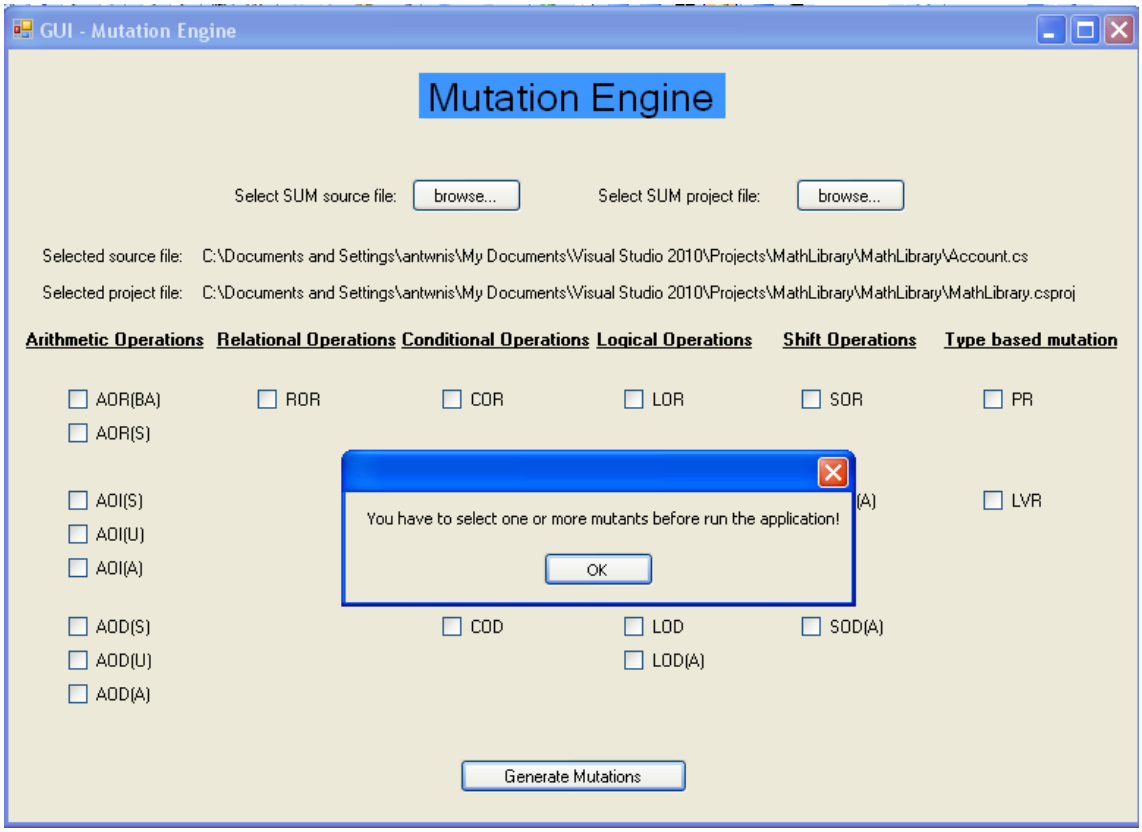
Σχετικές οθόνες παρουσίασης μηνυμάτων:



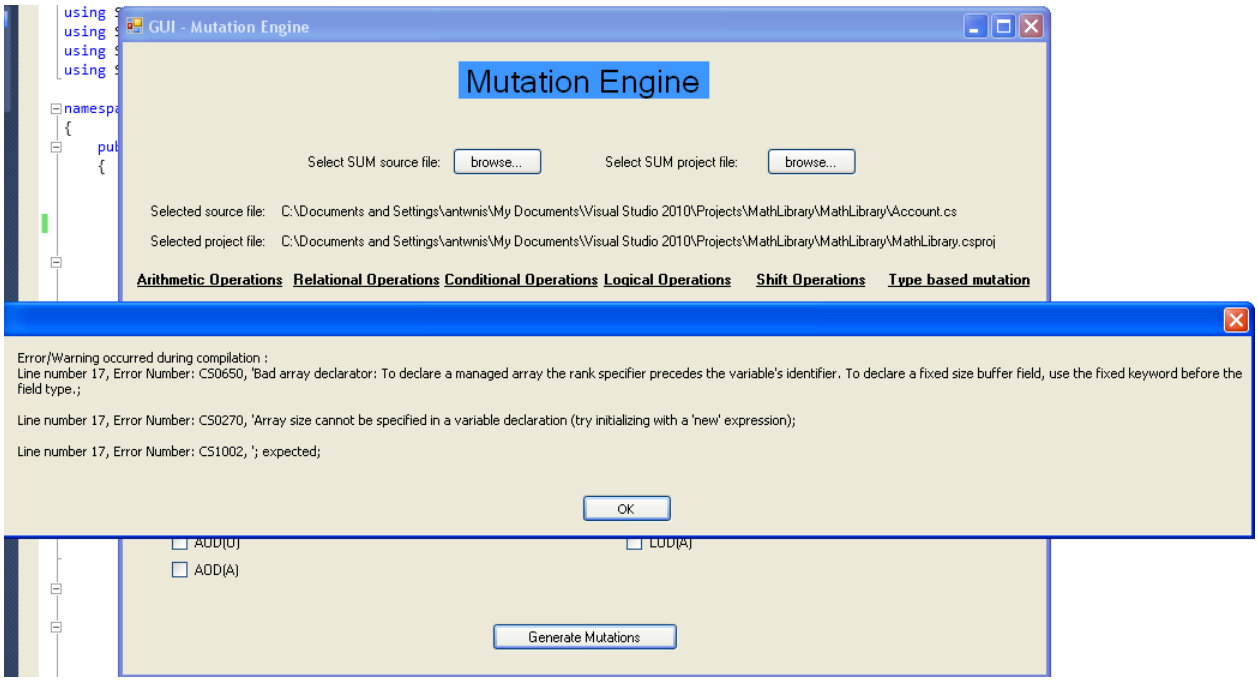
Εικόνα 5.3 – Απουσία μονοπατιού του υπό μετάλλαξη πηγαίου κώδικα



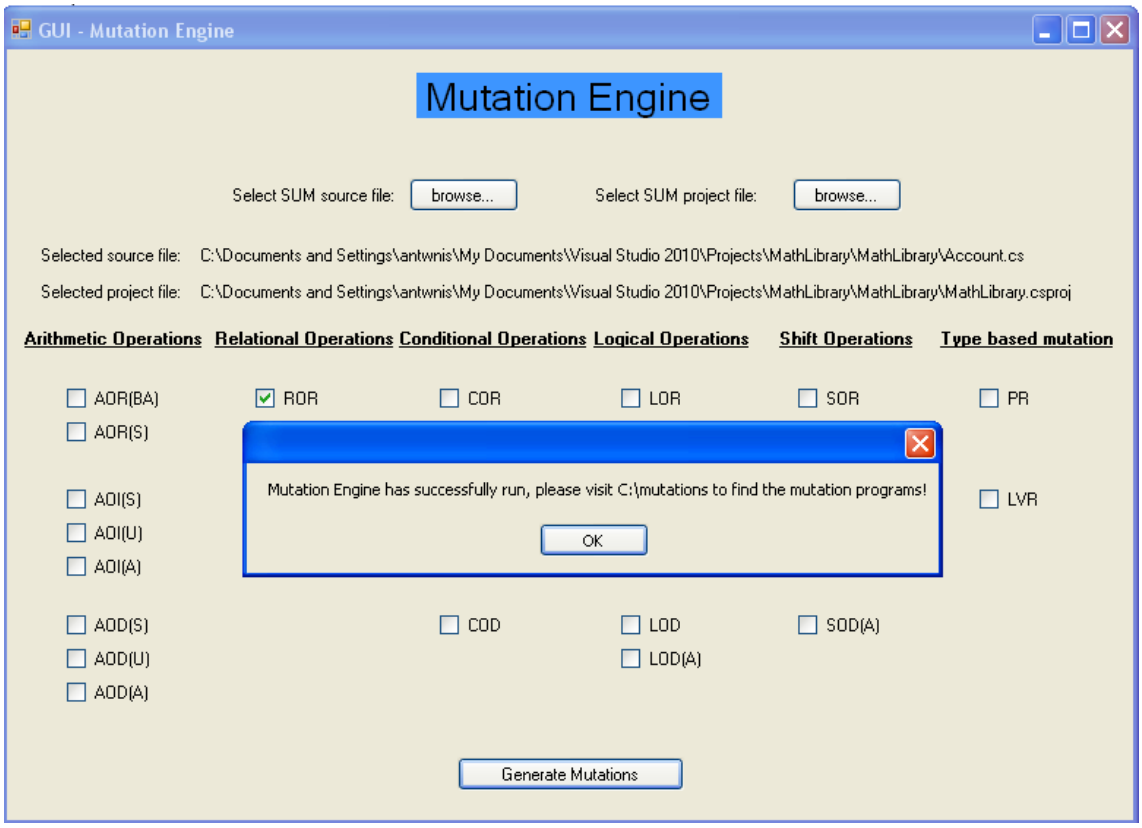
Εικόνα 5.4 – Απουσία μονοπατιού του project file του υπό μετάλλαξη πηγαίου κώδικα



Εικόνα 5.5 – Απουσία επιλογής τουλάχιστον ενός τελεστή μετάλλαξης



Εικόνα 5.6 – Συντακτικό σφάλμα στον πηγαίο κώδικα του υπό μετάλλαξη συστήματος λογισμικού



Εικόνα 5.7 – Επιτυχημένη παραγωγή μετάλλαξης του υπό μετάλλαξη συστήματος λογισμικού

Κεφάλαιο 6

Πειράματα

6.1 Εισαγωγή

Σε αυτό το κεφάλαιο θα παρουσιαστούν διάφορα παραδείγματα τα οποία έχουν εφαρμοστεί στην μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα που έχει υλοποιηθεί σε αυτή την εργασία. Μέσω αυτών των παραδειγμάτων θα αξιολογηθεί η εγκυρότητα του εργαλείου. Επίσης θα αναλυθεί ένα από τα παραδείγματα βάσει της καινοτόμου ιδέας που έχει προταθεί στο προηγούμενο κεφάλαιο έτσι ώστε να διαφανεί η επίδρασή της στην διαδικασία παραγωγής μεταλλάξεων.

6.2 Παραδείγματα

Σε αυτό το υποκεφάλαιο θα παρουσιάσουμε ενδεικτικά μέρη πηγαίου κώδικα τα οποία υπάρχουν στο σύνολο των προγραμμάτων που χρησιμοποιήθηκαν για την αξιολόγηση του εργαλείου και την επικύρωση της εγκυρότητας της λειτουργίας του, με σκοπό την παρουσίαση της λειτουργίας του. Θα παρουσιάζεται το αρχικό κομμάτι κώδικα καθώς και το σύνολο των μεταλλάξεων που έχουν δημιουργηθεί. Το κάθε παράδειγμα είναι ενδεικτικό του τρόπου λειτουργίας συγκεκριμένου τελεστή μετάλλαξης.

Παράδειγμα 1 – τελεστής μετάλλαξης AOR_{BA}

```
public int ToInt()
{

    return this.num / this.den;

}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 4 μεταλλαγμένα προγράμματα, αντικαθιστώντας τον τελεστή διαίρεσης με τους τελεστές πρόσθεσης, αφαίρεσης, πολλαπλασιασμού και ακέραιας διαίρεσης.

Παράδειγμα 2 – τελεστής μετάλλαξης AOR_S

```
public int nextNumber()
{
    return ++a;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 3 μεταλλαγμένα προγράμματα αντικαθιστώντας την πράξη ++a, με τις πράξεις --a, a++ και a--.

Παράδειγμα 3 – τελεστής μετάλλαξης AOI_S

```
public int multi()
{
    return (a * b);
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 4 μεταλλαγμένα προγράμματα εισάγοντας στην πράξη (a * b) τους τελεστές ++ και --. Δηλαδή η πράξη (a * b) έχει μεταλλαχθεί σε (++a * b), (--a * b), (a * ++b) και (a * --b). Οι τελεστές δεν εφαρμόζονται στο πίσω μέρος των μεταβλητών γιατί δεν θα έχουν άμεσο αντίκτυπο στη γραμμή αυτή, με αποτέλεσμα η περίπτωση αυτή να καλύπτεται από την μετάλλαξη της επόμενης παρουσίας της ίδιας μεταβλητής.

Παράδειγμα 4 – τελεστής μετάλλαξης AOI_U

```
public int multi()
{
    return (a * b);
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 2 μεταλλαγμένα προγράμματα εισάγοντας στην πράξη (a * b) τον αρνητικό τελεστή – μπροστά από τις μεταβλητές a και b.

Παράδειγμα 5 – τελεστής μετάλλαξης AOI_A

```
public void setA(int num1)
{
    a = num1;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 5 μεταλλαγμένα προγράμματα εισάγοντας στην ισότητα (a = num1) τους τελεστές πρόσθεσης, αφαίρεσης, διαίρεσης, πολλαπλασιασμού και της ακέραιας διαίρεσης. Να σημειωθεί πως η μεταβλητή a ελέγχεται από το εργαλείο εάν είναι αρχικοποιημένη πριν μεταλλαχθεί, έτσι ώστε να μην δημιουργηθεί μεταλλαγμένο πρόγραμμα με σφάλμα.

Παράδειγμα 6 – τελεστής μετάλλαξης AOD_S

```
public int nextNumber()
{
    return ++a;
}

public int prevNumber()
{
    --a;
    return a;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον τελεστή ++ από την εντολή return ++a. Ο τελεστής –a, δεν αφαιρέθηκε από την μεταβλητή a γιατί τότε το μεταλλαγμένο πρόγραμμα δεν θα ήταν συντακτικά ορθό.

Παράδειγμα 7 – τελεστής μετάλλαξης AOD_U

```
public int check()
{
    if (++a < 10)
        return ~a;
    else
        return -a;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον αρνητικό τελεστή – που βρίσκεται πριν τη μεταβλητή a.

Παράδειγμα 8 – τελεστής μετάλλαξης AOD_A

```
int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }

            int tri = 0;

            if (i == j)
            {
                tri += 1;
            }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον τελεστή πρόσθεσης από την εντολή
tri +=1.

Παράδειγμα 9 – τελεστής μετάλλαξης ROR

```
int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }

            int tri = 0;

            if (i == j)
            {
                tri += 1;
            }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 20 μεταλλαγμένα προγράμματα κάνοντας αλλαγές στους σχεσιακούς τελεστές <=, με τους σχεσιακούς τελεστές <,>,>=,= = και !=, καθώς και στη σχέση = = με τους τελεστές <,>,>=,<= και !=.

Παράδειγμα 10 – τελεστής μετάλλαξης COR

```
int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 2 μεταλλαγμένα προγράμματα κάνοντας αλλαγές στους τελεστές συνθήκης ||, ανταλλάζοντας τους με τους τελεστές &&.

Παράδειγμα 11 – τελεστής μετάλλαξης COI

```
int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 5 μεταλλαγμένα προγράμματα εισάγοντας τον τελεστή άρνησης ! στις εξής συνθήκες

- i <= 0
- j <= 0
- k <= 0
- ((i <= 0) || (j <= 0))
- ((j <= 0) || (k <= 0))

Παράδειγμα 12 – τελεστής μετάλλαξης COD

```
        public int min()
        {
            if (!(a < b))
                return b;
            else
                return a;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον τελεστή άρνησης ! από τη συνθήκη a < b.

Παράδειγμα 13 – τελεστής μετάλλαξης LOR

```
        public bool getLogicalAnd(bool b1, bool b2)
        {
            return b1 & b2;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 2 μεταλλαγμένα προγράμματα αντικαθιστώντας την λογική πράξη & με τις λογικές πράξεις | και ^.

Παράδειγμα 14 – τελεστής μετάλλαξης LOI

```
        public int max()
        {
            if (a > b)
                return a;
            else
                return b;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 4 μεταλλαγμένα προγράμματα εισάγοντας τον τελεστή ~ μπροστά από τις μεταβλητές a και b. Για να ενσωματωθεί ο πιο πάνω τελεστής μπροστά από μια μεταβλητή , πρέπει η μεταβλητή να είναι τύπου int ή long.

Παράδειγμα 15 – τελεστής μετάλλαξης LOI_A

```
public bool test()
        {
            bool a;
            bool b;

            a = true;
            b = !a;

            b = a ^ false;

            a = b & a;

            a |= !b;

            return b;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 6 μεταλλαγμένα προγράμματα εισάγοντας τους λογικούς τελεστές &,| και ^ στις εξισώσεις b = a ^ false και a= b & a, πριν από τον τελεστή ισότητας =. Οι εξισώσεις a = true και b=!a δεν έχουν μεταλλαχθεί γιατί οι μεταβλητές a και b σε εκείνο το στάδιο δεν έχουν τιμές, με αποτέλεσμα να προκαλούν συντακτικό σφάλμα.

Παράδειγμα 16 – τελεστής μετάλλαξης LOD

```
        public int check()
        {
            if (++a < 10)
                return ~a;
            else
                return -a;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον τελεστή ~.

Παράδειγμα 17 – τελεστής μετάλλαξης LOD_A

```
public bool test()
        {
            bool a;
            bool b;

            a = true;
            b = !a;

            b = a ^ false;

            a = b & a;

            a |= !b;

            return b;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα διαγράφοντας τον τελεστή |.

Παράδειγμα 18 – τελεστής μετάλλαξης SOR

```
        public int shift()
        {
            a <<= b;
            return a;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα αντικαθιστώντας τον τελεστή << με τον τελεστή >>.

Παράδειγμα 19 – τελεστής μετάλλαξης SOI_A

```
public int abs()
{
    if (a < 0)
        a = a * -1;

    return a;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 2 μεταλλαγμένα προγράμματα εισάγοντας τους τελεστές >> και << πριν από τον τελεστή εξίσωσης.

Παράδειγμα 20 – τελεστής μετάλλαξης SOD_A

```
public int shift()
{
    a <<= b;
    return a;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 1 μεταλλαγμένο πρόγραμμα αφαιρώντας τον τελεστή <<.

Παράδειγμα 21 – τελεστής μετάλλαξης PR

```
ManageStrings(String var1, String var2)
{
    str1 = var1;
    str2 = var2;
}
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 3 μεταλλαγμένα προγράμματα, ανταλλάζοντας τις παραμέτρους var1 και var2 μεταξύ τους. Μία φορά στη δήλωση του contractor κάνοντας την ManageStrings(String var2, String var1) και ακολούθως ανταλλάζοντας τις μεταβλητές var1 και var2,

Παράδειγμα 22 – τελεστής μετάλλαξης LVR

```
public bool test()
        {
            bool a;
            bool b;

            a = true;
            b = !a;

            b = a ^ false;

            a = b & a;

            a |= !b;

            return b;
        }
```

Για το πιο πάνω κομμάτι κώδικα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα έχει δημιουργήσει 3 μεταλλαγμένα προγράμματα, ανταλλάζοντας τις μεταβλητές a και b. Οι δύο μεταβλητές δεν ανταλλάζονται στις γραμμές δήλωσής τους.

Από τα πιο πάνω παραδείγματα μπορεί κάποιος να κατανοήσει πως οι διάφορες μεταλλάξεις σε ένα κομμάτι κώδικα μπορούν να εντοπίσουν πιθανό λάθος. Το λάθος φυσικά πρέπει να εμπίπτει στην εμβέλεια των μεταλλάξεων, γι'αυτό όσο πιο πολλούς τελεστές έχουμε τόσο καλύτερα αποτελέσματα αναμένουμε.

6.3 Ανάλυση εφαρμογής καινοτόμου ιδέας

Σε αυτό το υποκεφάλαιο θα χρησιμοποιήσουμε ένα παράδειγμα πηγαίου κώδικα το οποίο έχει επεξεργαστεί η μηχανή παραγωγής μεταλλάξεων και θα συγκρίνουμε τα αληθινά αποτελέσματα που έχει δημιουργήσει το εργαλείο με τα αποτελέσματα της καινοτόμου ιδέας που έχει προταθεί σε προηγούμενο κεφάλαιο. Για τα αποτελέσματα συμπεριλαμβανομένου της καινοτόμου ιδέας, έχει χρησιμοποιηθεί ο στατικός αναλυτής που παρέχεται από το Visual Studio 2010, ο οποίος ειδικεύεται στον έλεγχο των συμβολαίων πηγαίου κώδικα (Code Contracts). Πιο συγκεκριμένα, για κάθε μεταλλαγμένο πρόγραμμα, φορτωνόταν στην πλατφόρμα Visual Studio 2010 η οποία ρυθμίστηκε στο να εκτελεί στατικό έλεγχο και γινόταν «build» ο κώδικας. Η πλατφόρμα αναλόγως εάν παραβιάζονταν τα προαπαιτούμενα (pre-conditions), αναμενόμενα (post-conditions) ή και τις σταθερές (invariants), ενημέρωνε τον χρήστη

μέσω κάποιων μηνυμάτων. Επιπλέον η πλατφόρμα πρότεινε νέα προαπαιτούμενα (pre-conditions), αναμενόμενα (post-conditions) ή και σταθερές (invariants) με σκοπό να διορθωθούν εάν και εφόσον ο χρήστης θεωρήσει πως αυτά εμπεριέχουν κάποιο λάθος, δυνατότητα ευπρόσδεκτη για κάθε χρήστη ο οποίος ελέγχει ένα λογισμικό σύστημα,.

```
class CompareParadigm
    {
        int num;
        int den;

        public CompareParadigm(int numerator, int denominator)
        {
            Contract.Requires(0 < denominator);
            Contract.Requires(0 <= numerator);
            Contract.Requires(numerator > denominator);

            this.num = numerator;
            this.den = denominator;
        }

        [ContractInvariantMethod]
        private void ObjectInvariant()
        {
            Contract.Invariant(this.den > 0);
            Contract.Invariant(this.num >= 0);
        }

        public int ToInt()
        {
            Contract.Ensures(Contract.Result<int>() >= 0);
            return this.num / this.den;
        }
    }

    class Run
    {
        static void Main(string[] args)
        {
            int num1 = 0;
            int num2 = 0;

            num2 = 3;
            num1 = num2 + 2;

            CompareParadigm rational = new CompareParadigm(num1, num2);
        }
    }
```

Η μηχανή μετάλλαξης πηγαίου κώδικα για αυτό το παράδειγμα έχει δημιουργήσει 38 μεταλλαγμένα προγράμματα. Αναλυτικότερα:

Τελεστής	**Αριθμός μεταλλάξεων**
AOR_{BA}	8
AOI_S	10
AOI_U	6
LOI	6
PR	3
LVR	5
	Σύνολο: 38

Πίνακας 6.1 – Αποτελέσματα εκτέλεσης «απλού» mutation engine

Μετά την εκτέλεση της μεθοδολογίας ενσωμάτωσης της καινοτόμου ιδέας, τα μεταλλαγμένα προγράμματα έχουν μειωθεί σε 16. Αναλυτικότερα:

Τελεστής	**Αριθμός μεταλλάξεων**
AOR_{BA}	5
AOI_S	7
AOI_U	0
LOI	2
PR	2
LVR	0
	Σύνολο: 16

Πίνακας 6.2 – Αποτελέσματα εκτέλεσης «έξυπνου» mutation engine

Οι μεταλλάξεις οι οποίες δεν έγιναν αποδεχτές από τον στατικό αναλυτή είναι οι εξής:

- num2 = -3, εισαγωγή του αρνητικού τελεστή –

Παραβίαση του pre-condition (0 < denominator) και του invariant (this.den > 0).

- num1 = -num2 + 2, εισαγωγή του αρνητικού τελεστή –

Παραβίαση του pre-condition (0 < numerator) και του invariant (this.num >= 0).

- num1 = num2 + - 2, εισαγωγή του αρνητικού τελεστή –

Παραβίαση του pre-condition (numerator > denominator).

- num1 = num2 - 2, ανταλλαγή του συμβόλου + με το –

Παραβίαση του pre-condition (numerator > denominator).

- num1 = num2 / 2, ανταλλαγή του συμβόλου + με το /

Παραβίαση του pre-condition (numerator > denominator).

- num1 = num2 % 2, ανταλλαγή του συμβόλου + με το %

Παραβίαση του pre-condition (numerator > denominator).

- num2 = num2 + 2, ανταλλαγή της μεταβλητής num1 με τη μεταβλητή num2

Παραβίαση του pre-condition (numerator > denominator).

- CompareParadigm(num2, num2), ανταλλαγή της μεταβλητής num1 με τη μεταβλητή num2

Παραβίαση του pre-condition (numerator > denominator).

- CompareParadigm(num1, num1), ανταλλαγή της μεταβλητής num2 με τη μεταβλητή num1

Παραβίαση του pre-condition (numerator > denominator).

- num1 = 3, ανταλλαγή της μεταβλητής num2 με τη μεταβλητή num1

Παραβίαση του pre-condition (0 < denominator) μιας και το num2 δεν παίρνει άλλη τιμή εκτός κατά την αρχικοποίησή του.

- num1 = num1 + 2, ανταλλαγή της μεταβλητής num2 με τη μεταβλητή num1

Παραβίαση του pre-condition (numerator > denominator).

- public CompareParadigm(int denominator, int numerator) , ανταλλαγή της παραμέτρου numerator με τη παράμετρο denominator.

Παραβίαση του post-condition Contract.Result<int>() >= 0;

- this.den = --denominator

Παραβίαση του invariant (this.den > 0).

- return -- this.num / this.den

Παραβίαση του post-condition Contract.Result<int>() >= 0;

- return this.num / -- this.den

Παραβίαση του post-condition Contract.Result<int>() >= 0;

- this.den = ~denominator

Παραβίαση του invariant (this.den > 0).

- return ~this.num / this.den

Παραβίαση του post-condition Contract.Result<int>() >= 0;

- return this.num / ~this.den

Παραβίαση του post-condition Contract.Result<int>() >= 0;

- num2 = ~3

Παραβίαση του pre-condition (0 < denominator) και του invariant (this.den > 0).

Συνοπτικά, η χρήση της καινοτόμου ιδέας, μείωσε κατά 58%, τα παραγόμενα μεταλλαγμένα προγράμματα, απαλείφοντας τις «αχρείαστες» μεταλλάξεις σε ένα μικρό και απλό σχετικά παράδειγμα. Σε τεχνικές εύρεσης σφαλμάτων πηγαίο κώδικα, η μείωση αυτή θα οδηγήσει σε καλύτερα αποτελέσματα τόσο αποτελεσματικά όσο και αποδοτικά.

6.4 Αποτελέσματα

Σε αυτό το υποκεφάλαιο θα παρουσιασθούν τα αποτελέσματα του εργαλείου μέσω κάποιων πειραμάτων που έχουν γίνει. Στόχος του εργαλείου είναι να ενσωματώσει ατομικά λάθη στον κώδικα είτε με σκοπό την χρησιμοποίησή τους για την αποτίμηση του βαθμού καταλληλότητας των περιπτώσεων δοκιμής είτε για τον εντοπισμό

σφαλμάτων στον πηγαίο κώδικα Για τα πειράματα χρησιμοποιήθηκαν προγράμματα τα οποία έχουν υλοποιηθεί κατά την διάρκεια της εργασίας αυτής (Appendix).

6.4.1 Πειράματα παραγωγής μεταλλάξεων σε πηγαίο κώδικα

Η συμπεριφορά του εργαλείου ως προς τα προγράμματα που χρησιμοποιήθηκαν αποτιμάται στον πιο κάτω πίνακα αποτελεσμάτων. Για κάθε πρόγραμμα παρουσιάζεται ο αριθμός των παραγόμενων προγραμμάτων λόγω του ανάλογου τελεστή μετάλλαξης. Για καλύτερη αποτίμηση της εικόνας της λειτουργίας του εργαλείου παρουσιάζονται και οι γραφικές παραστάσεις για κάθε ομάδα τελεστών μετάλλαξης.

	AOR(BA)	AOR(S)	AOI(S)	AOI(U)	AOI(A)	AOD(S)	AOD(U)	AOD(A)
Account	8	0	12	6	0	0	0	2
Calculator	20	9	58	28	25	2	2	0
Manage Strings	8	0	16	8	0	0	0	0
Rational	4	0	8	4	15	0	1	0
Search	16	0	26	14	10	0	1	0
Triangle Cl	36	0	94	55	60	0	0	3
Manage Bool	0	0	0	0	0	0	0	0

Πίνακας 6.3 – Αποτελέσματα εκτέλεσης αριθμητικών πράξεων

	ROR
Account	0
Calculator	20
Manage Strings	0
Rational	0
Search	15
Triangle Cl	105
Manage Bool	0

Πίνακας 6.4 – Αποτελέσματα εκτέλεσης σχεσιακών πράξεων

	COR	COI	COD
Account	0	2	0
Calculator	0	4	1
Manage Strings	0	0	0
Rational	0	0	0
Search	0	3	0
Triangle Cl	7	28	0
Manage Bool	0	19	2

Πίνακας 6.5 – Πίνακας Αποτελεσμάτων Πράξεων Συνθηκών

	LOR	LOI	LOI(A)	LOD	LOD(A)
Account	0	0	0	0	0
Calculator	0	28	0	1	0
Manage Strings	0	8	0	0	0
Rational	0	4	0	0	0
Search	0	15	0	0	0
Triangle Cl	0	55	0	0	0
Manage Bool	12	0	6	0	1

Πίνακας 6.6 – Αποτελέσματα Λογικών Πράξεων

	SOR	SOI(A)	SOD(A)
Account	0	0	0
Calculator	1	8	1
Manage Strings	0	0	0
Rational	0	4	0
Search	1	4	0
Triangle Cl	0	24	0
Manage Bool	0	0	0

Πίνακας 6.7 – Πίνακας Αποτελεσμάτων Shift Πράξεων

	PR	LVR
Account	3	0
Calculator	3	0
Manage Strings	3	8
Rational	4	0
Search	0	33
Triangle Cl	93	0
Manage Bool	17	11

Πίνακας 6.8 – Πίνακας Αποτελεσμάτων Μεταλλάξεων Μεταβλητών

6.4.2 Πείραμα αποτίμησης βαθμού καταλληλότητας περιπτώσεων δοκιμής

Σε αυτό το πείραμα, παρουσιάζεται παράδειγμα εφαρμογής της μηχανής παραγωγής περιπτώσεων δοκιμής ως προς την αποτίμηση του βαθμού καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής. Η μεθοδολογία που ακολουθήθηκε είναι η εξής:

Στόχος της μηχανής είναι να δημιουργήσει τέτοια μεταλλαγμένα προγράμματα τα οποία θα οδηγήσουν το σύνολο από περιπτώσεις δοκιμής να αναγνωρίσουν την ύπαρξη διαφορετικότητας μεταξύ των μεταλλαγμένων προγραμμάτων και του αρχικού.

Triangle Classification

```
int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }
            int tri = 0;

            if (i == j)
              tri += 1;

            if (i == k)
              tri += 2;

            if (j == k)
               tri += 3;

            if (tri == 0)
            {
                if ((i + j == k) || (j + k <= i) || (i + k <= j))
                    tri = 4;
                else
                    tri = 1;
            }
            else
            {
                if (tri > 3)
                    tri = 3;
                else
                {
                    if ((tri == 1) && (i + j > k))
                        tri = 2;
                    else
                    {
                        if ((tri == 2) && (i + k > j))
                            tri = 2;
                        else
                        {
                            if ((tri == 3) && (j + k > i))
                                tri = 2;
                            else
                                tri = 4;
                        }
                    }
                }
            }

            return tri;
        }
    }
```

#	i	j	k
1	2	2	2
2	0	1	2
3	3	3	1
4	3	4	2

Πίνακας 6.9 - υπό έλεγχο σύνολο προπτώσεων δοκιμής

Εκ πρώτης όψεως οι πιο πάνω περιπτώσεις δοκιμής, εμφανίζονται ως κατάλληλες για το πρόγραμμα triangle classification γιατί καλύπτουν και τις τέσσερις περιπτώσεις πιθανόν αποτελεσμάτων που μπορεί να δημιουργήσει. Η πρώτη περίπτωση δοκιμής οδηγεί στο αποτέλεσμα ότι το τρίγωνο είναι ισόπλευρο. Η δεύτερη περίπτωση δοκιμής οδηγεί στο αποτέλεσμα πως οι τρεις πλευρές δεν συντελούν τρίγωνο. Η τρίτη περίπτωση δοκιμής οδηγεί στο αποτέλεσμα ότι το τρίγωνο είναι ισοσκελές. Η τελευταία περίπτωση δοκιμής οδηγεί στο αποτέλεσμα ότι το τρίγωνο είναι σκαληνό.

Εάν η μηχανή παραγωγής μεταλλάξεων δημιουργήσει έστω και ένα πρόγραμμα το οποίο θα περιέχει ένα ατομικό σφάλμα και το σύνολο των περιπτώσεων δοκιμής δεν το αναγνωρίσει, δηλαδή έχει ακριβώς την ίδια συμπεριφορά με το αρχικό πρόγραμμα τότε το σύνολο αυτό κρίνεται ως ακατάλληλο για εύρεση σφαλμάτων με αποτέλεσμα να μην θεωρείται κατάλληλο για τον έλεγχο του προγράμματος. Αποτέλεσμα αυτού η μείωση του βαθμού καταλληλότητας που το χαρακτηρίζει. Όσα περισσότερα μεταλλαγμένα προγράμματα δεν αναγνωρισθούν ως διαφορετικά από το αρχικό τόσο χαμηλότερος θα είναι ο βαθμός καταλληλότητας.

Στο πιο πάνω παράδειγμα, και μόλις στην έβδομη μετάλλαξη από τις εκατό πέντε που έχουν πραγματοποιηθεί, παρατηρείται η μη αναγνώριση της διαφορετικότητας του μεταλλαγμένου προγράμματος με το αρχικό. Η μετάλλαξη που έχει πραγματοποιηθεί είναι η πιο κάτω (στην $3^{η}$ γραμμή του προγράμματος):

```
if ((i <= 0) || (j < 0) || (k <= 0))
```

Εάν τοποθετήσουμε το σύνολο των περιπτώσεων δοκιμής στο μεταλλαγμένο αυτό πρόγραμμα θα παρατηρήσουμε ότι το αποτέλεσμα είναι το ίδιο με αυτό του αρχικού προγράμματος. Ο λόγος είναι ότι οι περιπτώσεις δοκιμής δεν διαπερνούν αυτή την ροή έτσι ώστε να αναγνωριστεί η διαφορετικότητα του μεταλλαγμένου προγράμματος. Αποτέλεσμα αυτού, η αναγνώριση μέσω της τεχνικής αυτής, της ανεπάρκειας του συνόλου περιπτώσεων δοκιμής να αναγνωρίσει όλα τα ενσωματωμένα σφάλματα.

Έτσι η μηχανή παραγωγής μεταλλάξεων μπορεί να βοηθήσει στην αποτίμηση του βαθμού καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής.

6.4.3 Πειράματα εύρεσης σφαλμάτων σε πηγαίο κώδικα

Σε αυτή τη σειρά πειραμάτων, παρουσιάζονται παραδείγματα εφαρμογής της μηχανής παραγωγής περιπτώσεων δοκιμής ως προς την εύρεση σφαλμάτων σε πηγαίο κώδικα. Η μεθοδολογία που ακολουθήθηκε είναι η εξής:
Αρχικά ενσωματώθηκαν σφάλματα στον κώδικα τα οποία οδηγούν το πρόγραμμα σε μη ομαλή λειτουργία σε συγκεκριμένες ροές ελέγχου. Το αρχικό πρόγραμμα είναι το ορθό. Στόχος της μηχανής είναι να δημιουργήσει τέτοια μεταλλαγμένα προγράμματα τα οποία θα οδηγήσουν στην εύρεση των σφαλμάτων που έχουν ενσωματωθεί.

Πρόγραμμα εύρεσης μεγαλύτερου αριθμού μεταξύ τεσσάρων ακέραιων αριθμών

```
public class FindMax
    {
        public int getMax(int num1, int num2, int num3, int num4)
        {
            int max = 0;
            if (num1 > num2){
                max = num1;
            }
            else
            {max = num3;
            }

            if (max < num3)
            {
                max = num3;
                if (max > num4)
                {
                    max = num3;
                }
            }
            else
            {
                if (max < num4)
                {
                    max = num4;
                }
            }
            return -max;
        }
    }
```

Στο πιο πάνω πρόγραμμα έχουν ενσωματωθεί τρία σφάλματα (με κόκκινο χρώμα). Πρώτο σφάλμα, στην θέση του num2 τοποθετήθηκε η μεταβλητή num3. Δεύτερο σφάλμα η τοποθέτηση του τελεστή >, στη θέση του < στην γραμμή if (max < num4). Τελευταίο σφάλμα η επιστροφή του μέγιστου αριθμού, αλλοιώθηκε με την εισαγωγή του αρνητικού τελεστή -.

Μετά την χρήση της μηχανής παραγωγής μεταλλάξεων σε πηγαίο κώδικα οι τελεστές ROR, PR και AOD_U έχουν ανακαλύψει τα σφάλματα τα οποία έχουν ενσωματωθεί στο αρχικό πρόγραμμα.

Ανάλυση εύρεσης σφαλμάτων

Τελεστής ROR:

Ο τελεστής σχεσιακών πράξεων έχει μεταλλάξει πέντε φορές την γραμμή με το σφάλμα. Έχει μετατρέψει τον τελεστή > σε <,>=,<=,= =,!=. Παρατηρούμε πως η μετάλλαξη με τον τελεστή < είναι και αυτή που αναγνωρίζει το σφάλμα.

```
if (max > num4)
```
→
```
if (max < num4)
```

Τελεστής PR:

Ο τελεστής ανταλλαγής παραμέτρων δημιουργεί τόσα προγράμματα όσα και οι πιθανοί συνδυασμοί των τεσσάρων παραμέτρων (num1, num2, num3, num4). Στον συνδυασμό ανταλλαγής της μεταβλητής num3 με το num2 παρατηρούμε ότι εκτός των άλλων, αντικαθιστά την μεταβλητή num3 με την μεταβλητή num2 στην γραμμή όπου όντως υπάρχει το σφάλμα. Αποτέλεσμα αυτού η εύρεση του σφάλματος.

```
max = num3;
```
→
```
max = num2;
```

Τελεστής AOD_U:

Ο τελεστής διαγραφής unary αριθμητικών πράξεων, αφαιρεί το αρνητικό πρόσημου -. Η μόνη παρουσία του προσήμου αυτού είναι και η γραμμή όπου περιέχει το σφάλμα.

```
return -max;
```
→
```
return max;
```

Πρόγραμμα διαίρεσης δύο ακέραιων αριθμών συμπεριλαμβανομένου προδιαγραφών μέσω των Code Contracts

```
class CompareParadigm
{
    int num;
    int den;

    public CompareParadigm(int numerator, int denominator)
    {
        Contract.Requires(0 < denominator);
        Contract.Requires(0 <= numerator);
        Contract.Requires(numerator > denominator);
        this.num += numerator;
        this.den = denominator;
    }

    [ContractInvariantMethod]
    private void ObjectInvariant()
    {
        Contract.Invariant(this.den > 0);
        Contract.Invariant(this.num >= 0);
    }

    public int ToInt()
    {
        Contract.Ensures(Contract.Result<int>() >= 0);
        return this.num * this.den;
    }
}
```

Στο πιο πάνω πρόγραμμα έχουν ενσωματωθεί δύο σφάλματα (με κόκκινο χρώμα). Τα παρακάτω σφάλματα δεν εντοπίζονται από τον στατικό αναλυτή των Code Contracts μιας και δεν καταπατούν τις καταγραμμένες προδιαγραφές. Πρώτο σφάλμα, η εισαγωγή του τελεστή πρόσθεσης + στην ισότητα this.num = numerator. Δεύτερο σφάλμα η ανταλλαγή του τελεστή διαίρεσης / με τον τελεστή πολλαπλασιασμού *.

Μετά την χρήση της μηχανής παραγωγής μεταλλάξεων σε πηγαίο κώδικα οι τελεστές AORBA και AODA εντοπίζουν τα δύο σφάλματα τα οποία έχουν ενσωματωθεί στο αρχικό πρόγραμμα.

Ανάλυση εύρεσης σφαλμάτων

Τελεστής AORBA:

Ο τελεστής AORBA αντικαθιστά τις αριθμητικές πράξεις με όλες τις υπόλοιπες. Έτσι θα μεταλλάξει τις δύο γραμμές που περιέχουν το σφάλμα, μιας και έχουν αριθμητικές πράξεις += και *. Την πρώτη γραμμή θα την μεταλλάξει σε -=,*=,/=,%= αλλά όπως θα διαφανεί αργότερα δεν περιέχει κάποιο σφάλμα αυτής της μορφής η γραμμή. Αντιθέτως, η γραμμή όπου περιέχει τον πολλαπλασιασμό θα μεταλλαχθεί σε +.-,/,% με αποτέλεσμα η μετάλλαξη με τον τελεστή διαίρεσης / να εντοπίζει το ενσωματωμένο σφάλμα.

```
return this.num * this.den;
```
→
```
return this.num / this.den;
```

Τελεστής AODA:

Ο τελεστής διαγράφει της αριθμητικές πράξεις η οποίες βρίσκονται σε εξίσωση τύπου +=,-=,*=,/= και%=. Αποτέλεσμα αυτού την διαγραφή του τελεστή πρόσθεσης + καθώς και τον εντοπισμό του συγκεκριμένου λάθους.

```
this.num += numerator;
```
→
```
this.num = numerator;
```

Γενικότερα, η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα, μπορεί να βοηθήσει αρκετά στην εύρεση σφαλμάτων τα οποία εμπίπτουν στις δυνατότητες των τελεστών μετάλλαξης που παρέχει. Δηλαδή, αριθμητικές πράξεις (προσθαφαιρέσεις και ανταλλαγές), σχεσιακές πράξεις (ανταλλαγές), πράξεις συνθηκών (προσθαφαιρέσεις και ανταλλαγές), λογικές πράξεις (προσθαφαιρέσεις και ανταλλαγές), shift πράξεις (προσθαφαιρέσεις και ανταλλαγές) καθώς και ανταλλαγές μεταξύ μεταβλητών ή και παραμέτρων ιδίου τύπου οι οποίες βρίσκονται στην ίδια εμβέλεια.

Με κάποιο αλγόριθμο, για παράδειγμα γενετικό, ο οποίος θα χρησιμοποιεί unit testing μέσω ενός συνόλου από περιπτώσεις δοκιμής καθώς και τον επιθυμητών αποτελεσμάτων για την επιβεβαίωση της ανεύρεσης του λάθους, μπορούν να αξιοποιηθούν οι τρεις αυτές ευρέσεις και να συνδυαστούν έτσι ώστε να επιστρέψουν το

ορθό πρόγραμμα. Η κατάλληλη επιλογή τελεστών μετάλλαξης επηρεάζουν την αποδοτικότητα του πιο πάνω αλγορίθμου. Αυτό είναι ένα θέμα το οποίο μπορεί να μελετηθεί σε επόμενη έρευνα.

Κεφάλαιο 7

Συμπεράσματα και Μελλοντική Εργασία

7.1 Εισαγωγή

Στο κεφάλαιο αυτό θα αναφερθούν κάποιοι περιορισμοί που υπάρχουν στην εφαρμογή καθώς και τα συμπεράσματα από την εργασία αυτή. Στο τέλος θα υπάρξουν κάποιες ιδέες και παροτρύνσεις έτσι ώστε να βελτιστοποιηθεί τόσο η υπάρχουσα υλοποίηση καθώς και παρόμοιες υλοποιήσεις (μελλοντικές μελέτες).

7.2 Περιορισμοί

Στην εργασία αυτή μελετήθηκε η πλατφόρμα Visual Studio 2010 καθώς και τα Code Contracts τα οποία είναι πρόσφατα προϊόντα, με συνέπεια να μην είναι ακόμη σταθερά. Και οι δύο τεχνολογίες έχουν κάποιο υπόβαθρο, το Visual Studio έχει το 2000 – 2005 – 2008 και τα συμβόλαια πηγαίου κώδικα (Code Contracts) έχουν τη γλώσσα SPEC#. Δεν παύει όμως ο κίνδυνος του εντοπισμού bugs, ειδικά των καινούργιων υπηρεσιών που παρέχουν. Στα θετικά συγκαταλέγονται η ευρεία χρήση τους και η συνεχής βελτίωσή τους από την κατασκευάστρια εταιρία.

Η μηχανή παραγωγής μεταλλαγμένων προγραμμάτων που υλοποιήθηκε καθώς και το component της στατικής ανάλυσης, χειρίζεται μόνο προγράμματα γραμμένα σε γλώσσα προγραμματισμού C#. Η δομή με την οποία υλοποιήθηκαν επιτρέπει την ενσωμάτωση και των άλλων γλωσσών που παρέχει η πλατφόρμα Visual Studio 2010.

Επίσης η μηχανή παραγωγής μεταλλαγμένων προγραμμάτων που υλοποιήθηκε, υποστηρίζει μόνο τους traditional method-based τελεστές. Λόγω της αντικειμενοτρεφούς φύσης των γλωσσών προγραμματισμού που παρέχει η πλατφόρμα Visual Studio 2010 μπορούν να υλοποιηθούν και οι τελεστές μετάλλαξης επιπέδου κλάσεων. Ο τελεστής COI, χρειάζεται την παρουσία αγκυλών για να λειτουργήσει για όλες τις περιπτώσεις, λόγω του αλγορίθμου τεμαχισμού που ακολουθεί.

Παρά τους πιο πάνω περιορισμούς, κανένας από αυτούς δεν αποτελεί αποτρεπτικό παράγοντα στην επαναχρησιμοποίηση των components που υλοποιήθηκαν αλλά και της μηχανής παραγωγής μεταλλάξεων σε πηγαίο κώδικα. Όλα αυτά μπορούν να βοηθήσουν σε επόμενες ερευνητικές μεθόδους ελέγχου λογισμικών συστημάτων.

7.3 Συμπεράσματα

Ο έλεγχος των λογισμικών είναι μία αρκετά σημαντική λειτουργία για την δημιουργία ορθών και αξιόπιστων λογισμικών τα οποία θα χρησιμοποιηθούν έτσι ώστε να μας διευκολύνο **ν** τη ζωή. Επίσης λόγω του ότι τα λογισμικά είναι πλέον αρκετά πολύπλοκα πρέπει να δημιουργηθούν κατάλληλες μέθοδοι ελέγχου έτσι ώστε να εκτελούν αυτοματοποιημένο έλεγχο.

Για τον έλεγχο ενός λογισμικού συστήματος, χρειάζεται ένα σύνολο από περιπτώσεις δοκιμής. Η εύρεση κατάλληλων περιπτώσεων δοκιμής δεν είναι καθόλου εύκολη διαδικασία. Μία τεχνική η οποία αναλύει τον βαθμό καταλληλότητας ενός συνόλου περιπτώσεων δοκιμής είναι αυτή του ελέγχου βάσει μεταλλάξεων σε πηγαίο κώδικα. Κύριο εργαλείο στην εφαρμογή της πιο πάνω τεχνικής, είναι η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα. Η μηχανή αυτή μπορεί να χρησιμοποιηθεί ως υποστηρικτικό εργαλείο και σε άλλους τομείς αναφορικά με τη διαδικασία ελέγχου λογισμικού συστήματος, όπως παραδείγματος χάρη τον εντοπισμό σφάλματος σε πηγαίο κώδικα. Με γνώμονα τη χρήση της, μπορεί να τύχει βελτιστοποίησης έτσι ώστε να βοηθηθεί στην αποδοτικότητα και την αποτελεσματικότητα του σκοπού για τον οποίο θα χρησιμοποιηθεί.

Για τη διεκπεραίωση οποιασδήποτε τεχνικής ελέγχου άσπρου κουτιού, επιβάλλεται η χρήση στατικής ανάλυσης κώδικα και η αναπαράσταση του σε τέτοια μορφή έτσι ώστε

να μπορεί εύκολα να κατανοηθεί αλλά και να διαπεραστεί για να ανακτηθούν οι απαιτούμενες πληροφορίες. Ένα από τα επιτεύγματα της εργασίας αυτής ήταν η υλοποίηση ενός στατικού αναλυτή ο οποίος χρησιμοποιεί λεξικογραφική ανάλυση, αναπαριστώντας τον πηγαίο κώδικα ως ένα Abstract Source Tree. Ακολούθως, διαπερνά το Abstract Source Tree και αποθηκεύει ένα σύνολο από χρήσιμες πληροφορίες σε διάφορες λίστες αναφορικά με τον τύπο της πληροφορίας. Ο στατικός αναλυτής μπορεί να χρησιμοποιηθεί σε αρκετές τεχνικών ελέγχου άσπρου κουτιού καθώς και σε τεχνικές αναπαράστασης κώδικα (Control Flow Graph, Data Dependence Graph).

Λόγω της φύσης της πλατφόρμας Visual Studio 2010, δηλαδή η ύπαρξη ενός solution, με αρκετά projects τα οποία περιέχουν αρκετές κλάσεις και εφαρμογές με GUI κτλ, κρίθηκε αναγκαία η δημιουργία ενός component το οποίο επιβεβαιώνει την ορθότητα του πηγαίου κώδικα. Το συστατικό αυτό, παίρνει ως είσοδο τον πηγαίο κώδικα καθώς και το project file, αναλύοντας τις απαραίτητες πληροφορίες που χρειάζεται για όλες τις αναφορές βιβλιοθηκών που χρησιμοποιεί.

Η υλοποιημένη μηχανή παραγωγής μεταλλάξεων εντολών σε πηγαίο κώδικα είναι ένα εργαλείο το οποίο μπορεί να χρησιμοποιηθεί για την επίτευξη της τεχνικής ελέγχου βάσει μεταλλάξεων (mutation testing). Επιπλέον υπάρχει η δυνατότητα να συνεργαστεί με άλλα εργαλεία με στόχο την εύρεση σφαλμάτων στον πηγαίο κώδικα. Η ενσωμάτωση της «καινοτόμου» ιδέας μπορεί να αυξήσει την αποτελεσματικότητα της μηχανής μειώνοντας τα «αχρείαστα» σε ορισμένες περιπτώσεις (διαδικασία εύρεσης σφαλμάτων) μεταλλαγμένα προγράμματα.

7.4 Μελλοντική Εργασία

Όπως όλες οι έρευνες έτσι και αυτή έχει περιθώρια βελτιστοποίησης. Μετά από αρκετή τριβή με το θέμα καλό είναι να παρατεθούν μερικά στοιχεία τα οποία θα βοηθήσουν επόμενες μελλοντικές εργασίες τόσο σε αυτή την εφαρμογή όσο και γενικότερα στο θέμα αυτό.

Τόσο η στατική ανάλυση όσο και η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα, χειρίζονται μόνο προγράμματα γραμμένα σε C#. Η επέκτασή τους θα

μπορούσε να τους δώσει μια επιπλέον δυναμική και ισχύ. Ο τρόπος με τον οποίο είναι υλοποιημένα, δίδει την δυνατότητα επέκτασης τους χωρίς πολλές αλλαγές.

Η μηχανή παραγωγής μεταλλάξεων σε πηγαίο κώδικα καλύπτει ένα σύνολο από traditional method based τελεστές. Οι τελεστές αυτοί θα μπορούσαν να εμπλουτιστούν με την εισαγωγή class based τελεστών οι οποίοι θα χρησιμοποιήσουν την αντικειμενοστρέφεια που υπάρχει στις γλώσσες προγραμματισμού που παρέχει η πλατφόρμα Visual Studio 2010. Ακόμη μπορούν να προστεθούν και άλλοι τελεστές βάσει των αναγκών της εφαρμογής όπου θα χρησιμοποιηθεί.

Οποιαδήποτε τεχνική ελέγχου άσπρου κουτιού μπορεί να χρησιμοποιήσει τα components τα οποία υλοποιήθηκαν και είναι επαναχρησιμοποιήσιμα. Επίσης μπορεί να συνδυαστούν με άλλες δυνατότητες της πλατφόρμας Visual Studio 2010 και να υλοποιηθούν νέα υβριδικά σύστημα ελέγχου λογισμικών συστημάτων. Για παράδειγμα, μπορεί να υλοποιηθεί ένα σύστημα ελέγχου βάσει μεταλλάξεων, με τον συνδυασμό των ήδη υλοποιημένων συστατικών καθώς και της μηχανής παραγωγής μεταλλάξεων σε πηγαίο κώδικα, μαζί με το εργαλείο PEX το οποίο εκτελεί unit testing και θα υλοποιούσε το τελευταίο κομμάτι της τεχνικής ελέγχου βάσει μεταλλάξεων στον πηγαίο κώδικα. Με αυτή την υβριδική λύση θα προσδίδαμε την δυνατότητα στου Visual Studio και στα εργαλεία ελέγχου του να παράγουν κατάλληλες περιπτώσεις δοκιμής, οι οποίες καλύπτουν όλα τα μέρη του πηγαίου κώδικα.

Επίσης το εργαλείο UModel το οποίο παρέχει όλα τα UML σχήματα, μπορεί να χρησιμοποιηθεί σε μελλοντικές έρευνες καθώς και άλλα εργαλεία τα οποία έχουν υλοποιηθεί. Επίσης δυνατότητες όπως η ανάλυση κώδικα, η κάλυψη κώδικα και γενικότερα τα εργαλεία ελέγχου που αναφέρθηκαν σε προηγούμενο κεφάλαιο ανοίγουν νέες προοπτικές για έρευνες.

Θα μπορούσε να δημιουργηθεί τεχνική στατικής ανάλυσης η οποία να μπορεί να ελέγχει ένα σύνολο συνθηκών, έτσι ώστε να εφαρμοστεί και στην πράξη η καινοτόμος ιδέα που έχει προταθεί, και να εμφανιστούν τα πλεονεκτήματά της. Μία ιδέα είναι η χρησιμοποίηση του στατικού αναλυτή που παρέχει το Visual Studio αναφορικά με τα συμβόλαια πηγαίου κώδικα (Code Contracts). Για να υλοποιηθεί αυτό όμως πρέπει να

γίνει μια πιο εις βάθος μελέτη αναφορικά με το πώς μπορεί κάποιος να χρησιμοποιήσει το εργαλείο αυτό μέσω μιας εφαρμογής. Σε συνδυασμό με την τεχνική εύρεσης σφάλματος σε πηγαίο κώδικα (fault localization) θα ήταν ένα αρκετά καλό εργαλείο για την εύρεση σφαλμάτων σε πηγαίο κώδικα, βοηθώντας σημαντικά στη διαδικασία ελέγχου και αποσφαλμάτωσης ενός λογισμικού συστήματος.

Βιβλιογραφία

[1] P. McMinn, *Search-based Software Test Data Generation: A Survey*, 14 (2), pp 105 – 156, 2004

[2] Richard Graham Hamlet, “Testing Programs with the Aid of a Compiler”, IEEE Transactions on Software Engineering, 3(4), p. 279-290, July 1977.

[3] Richard A. DeMillo and Richard J. Lipton and Frederick Gerald Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, IEEE Computer 11(4), p. 34-41, April 1978.

[4] Yu-Seung Ma, Jeff Offutt, “Description of Class Mutation Operators for Java”, November 2005.

[5] Yu-Seung Ma, Jeff Offutt, “Description of Method-level Mutation Operators for Java”, November 2005.

[6] “Mutation Testing Repository”, http://www.dcs.kcl.ac.uk/pg/jiayue/repository/ Accessed: December 2009.

[7] A. J. Offutt, G. Rothermel, and C. Zapf. An experimental evaluation of selective mutation. In International Conference on Software Engineering, pages 100–107, 1993.

[8] J. Offutt, Y.-S. Ma, and Y. R. Kwon, “MuJava : An Automated Class Mutation System”, Software Testing, Verification and Reliability, vol. 15, pp. 97-133, 2005.

[9] P. Hoschka, C. Huitema INRIA, Control flow graph analysis for automatic fast path implementation, Second IEEE workshop on the architecture and Implementation of high performance communication subsystems, 2004

[10] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren, The Program Dependence Graph and its Use in Optimization, In ACM Transactions on Programming Languages and Systems, ACM Press, 9, 319-349, 1987.

[11] “SharpCode”, http://www.icsharpcode.net/opensource/sd/, Accessed: December 2005.

[12] “Spec#”, http://research.microsoft.com/en-us/projects/specsharp/, Accessed: December 2009.

[13]”Code Contracts User Manual”, 2010, Microsoft Corporation, http://research.microsoft.com/en-us/projects/contracts/userdoc.pdf

[14] Visual Studio Test Tools, 2010 http://blogs.msdn.com/b/terryclancy/archive/2009/11/18/visual-studio-2010-test-tools-partner-integration-opportunities.aspx

[15] “Code Coverage”, http://msdn.microsoft.com/en-us/library/dd299398(VS.90).aspx, Accessed: March 2008.

[16] “Pex and Moles - Isolation and White box Unit Testing for .NET”, http://research.microsoft.com/en-us/projects/pex/, Accessed: January 2008

[17] “UModel - UML tool for software modeling and application development” , http://www.altova.com/umodel.html, Accessed: January 2010

[18] NUnit, 2006, http://www.nunit.org/

[19] Gallio, 2009, http://www.gallio.org/

[20] NCover, 2007, http://ncover.sourceforge.net/

[21] R. A. DeMillo and E. H. Spafford, “The Mothra software testing environment”, presented at The 11th NASA Software Engineering Laboratory Workshop, Goddard Space Center, 1986.

[22] J. Offutt, Y.-S. Ma, and Y. R. Kwon, “MuJava : An Automated Class Mutation System”, Software Testing, Verification and Reliability, vol. 15, pp. 97-133, 2005.

[23] MuClipse, 2003, http://muclipse.sourceforge.net/

[24] Jester, 2000 http://jester.sourceforge.net/

[25] Nester, 2004, http://nester.sourceforge.net/

[26] Yue Jia , “MILU : A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language”

[27] A. A. Sofokleous, A. S. Andreou, C. Schizas and G. Ioakim, *Extending and enhancing a basic program analysis system*, Proceedings of the IADIS International Conference, Applied Computing 2006 (AC2006), San Sebastian, Spain, 2006

[28] A.A. Sofokleous and A.S. Andreou, *Automatic, Evolutionary Test Data Generation for Dynamic Software Testing*, Journal of Systems and Software, (Elsevier Science: Amsterdam, Netherlands), accepted.

[29] A.A. Sofokleous and A.S. Andreou, 2007, *Batch-Optimistic Test-Cases Generation Using Genetic Algorithms*, Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence (ICTAI), Patra, Greece, October, pp. 157-164.

[30] A.S. Andreou, K.A. Economides and A.A. Sofokleous, 2007, *An automatic software test-data generation scheme based on data flow criteria and genetic algorithms*, Proceedings of the 7th IEEE International Conference on Computer and Information Technology, Fukushima, Japan, October, (IEEE Computer Society: Los Alamitos, CA, USA), pp 867-872

[31] Διπλωματική Εργασία Αναστάση Σοφοκλέους, *έλεγχος κώδικα λογισμικού – Αυτόματη δημιουργία γράφου ελέγχου ροής και περιπτώσεις ελέγχου μέσω αλγόριθμων βελτιστοποίησης*

[32] Διπλωματική Εργασία Κυριάκου Ιωάννου, Ιούνιος 2007, *Δημιουργία και γραφική απεικόνιση/διαχείριση γράφου εξάρτησης δεδομένων από κώδικα προγραμμάτων γραμμένα σε Java*

[33] Διπλωματική Εργασία Ρεββέκας Χριστοδούλου, Ιούνιος 2006, *Αυτοματοποιημένος έλεγχος λογισμικού με βελτιστοποίηση του κριτηρίου κάλυψης μονοπατιού*

[34] Διπλωματική Εργασία Αντώνη Κουρρά, Ιούνιος 2008, *Αυτόματη παραγωγή περιπτώσεων ελέγχου λογισμικού μέσω συμβολικής εκτέλεσης προγραμμάτων από γενετικούς αλγορίθμους.*

Appendix

Account.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics.Contracts;

namespace MathLibrary
{
    public class Account
    {
        public readonly bool _suppOver;
        public float _am;
        public float _overLim;

        #region Contract (Invariants)

        [ContractInvariantMethod]
        private void Invariants()
        {
            Contract.Invariant(this.Amount == this._am);
            Contract.Invariant(this.OverdraftLimit == this._overLim);
            Contract.Invariant(this.SupportsOverdraft == this._suppOver);

            Contract.Invariant(this._suppOver ? this._overLim > 0 : true);
            Contract.Invariant(!this._suppOver ? this._overLim == 0 : true);
            Contract.Invariant(this._overLim > -1);
            Contract.Invariant(this._am > -this._overLim);
        }

        #endregion

        public Account(float openingAmount, bool supportsOverdraft = false, float
overdraftLimit = 0)
        {
            #region Contract

            // Preconditions
            Contract.Requires(openingAmount > -1);
            Contract.Requires(overdraftLimit > -1);
            Contract.Requires(openingAmount > -overdraftLimit);
            Contract.Requires(supportsOverdraft ? overdraftLimit > 0 : true);
            Contract.Requires(!supportsOverdraft ? overdraftLimit == 0 : true);

            // Post Conditions
            Contract.Ensures(this._am == openingAmount);
            Contract.Ensures(this._suppOver == supportsOverdraft);
            Contract.Ensures(this._overLim == overdraftLimit);

            #endregion
```

```
            this._am = openingAmount;
            this._suppOver = supportsOverdraft;
            this._overLim = overdraftLimit;
        }

        public float Amount
        {
            get
            {
                Contract.Ensures(this.Amount == this._am);

                return this._am;
            }
        }


        public float OverdraftLimit
        {
            get
            {
                Contract.Ensures(this.OverdraftLimit == this._overLim);

                return this._overLim;
            }
        }


        public bool SupportsOverdraft
        {
            get
            {
                Contract.Ensures(this.SupportsOverdraft == this._suppOver);

                return this._suppOver;
            }
        }

        public void Debit(float debitAmount)
        {
            #region Contract

            // Preconditions (P)
            Contract.Requires(debitAmount > -1);
            Contract.Requires(this._am - debitAmount > -this._overLim);

            // Post Conditions (state changes) (Q)
            Contract.Ensures(this._am == Contract.OldValue<float>(this._am) -
debitAmount);

            // Post Conditions (R)
            Contract.Ensures(this._suppOver ==
Contract.OldValue<bool>(this._suppOver));
            Contract.Ensures(this._overLim ==
Contract.OldValue<float>(this._overLim));

            #endregion

            this._am -= debitAmount;
        }
```

```
        public void Credit(float creditAmount)
        {
            #region Contract

            // Preconiditions
            Contract.Requires(creditAmount > -1);
            Contract.Requires(this._am + creditAmount > -this._overLim);

            // Post Conditions
            Contract.Ensures(this._am == Contract.OldValue<float>(this._am) +
creditAmount);

            #endregion


            this._am += creditAmount;
        }
    }

}
```

Calculator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;


namespace MathLibrary
{
    //<summary>
    //<Class Linear
    //<=/summary>
    public class Calculator
    {
        ///<summary>
        ///<integer A
        ///</summary>
        public int a;
        ///<<summary>
        ///<integer B
        ///<</summary>
        public int b;

        ///<<summary>
        ///<string t
        ///<</summary>
        public Calculator()
        {

        }


        ///<<summary>
        ///<string t
        ///<</summary>
        public Calculator(int n1, int n2)
        {
            a = n1;
            b = n2;
        }

        ///<<summary>
        ///<set integer A
        ///<</summary>
        public void setA(int num1)
        {
            a = num1;
        }



        /// <summary>
        /// set integer B
        /// </summary>
        public void setB(int num2)
```

```
{
    b = num2;
}

/// <summary>
/// find maximum between a and b
/// </summary>
public int max()
{
    if (a > b)
        return a;
    else
        return b;
}

/// <summary>
/// find minimum between a and b
/// </summary>
public int min()
{
    if (!(a < b))
        return b;
    else
        return a;
}

/// <summary>
/// find plus between a and b
/// </summary>
public int plus()
{
    return (a + b);
}

/// <summary>
/// find minus between a and b
/// </summary>
public int minus()
{
    return (a - b);
}

/// <summary>
/// find multi between a and b
/// </summary>
public int multi()
{
    return (a * b);
}

/// <summary>
/// find div between a and b
/// </summary>
public int div()
{
    int result = a / b;

    return result;
}

/// <summary>
```

```
        /// find absolute between a and b
        /// </summary>
        public int abs()
        {
            if (a < 0)
                a = a * -1;

            return a;
        }

        /// <summary>
        /// return shift result between a and b
        /// </summary>
        public int shift()
        {
            a <<= b;
            return a;
        }

        /// <summary>
        /// return next number
        /// </summary>
        public int nextNumber()
        {
            return ++a;
        }

        /// <summary>
        /// return previous number
        /// </summary>
        public int prevNumber()
        {
            --a;
            return a;
        }

        /// <summary>
        /// return check value
        /// </summary>
        public int check()
        {
            if (++a < 10)
                return ~a;
            else
                return -a;
        }
    }
}
```

ManageBoolean.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathLibrary
{
    class ManageBoolean
    {
        public bool getLogicalAnd(bool b1, bool b2)
        {
            return b1 & b2;
        }

        public bool getLogicalOr(bool b1, bool b2, bool b3)
        {
            return b1 | b2;
        }

        public bool getLogicalXOr(bool b1, bool b2, bool b3)
        {
            return b1 ^ b2;
        }

        public bool test()
        {
            bool a;
            bool b;

            a = true;
            b = !a;

            b = a ^ false;

            a = b & a;

            a |= !b;

            return b;
        }
    }
}
```

ManageStrings.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathLibrary
{
    class ManageStrings
    {
        String str1;
        String str2;

        ManageStrings(String var1, String var2)
        {
            str1 = var1;
            str2 = var2;
        }

        public string Conc()
        {
            string newString;

            newString = str1 + str2;

            return newString;
        }

        public int Diff()
        {
            int length1;
            int length2;

            length1 = str1.Length;

            length2 = str2.Length;

            return length1 - length2;
        }

        public int Add()
        {
            int length1;
            int length2;

            length1 = str1.Length;

            length2 = str2.Length;

            return length1 + length2;
        }
    }
}
```

Rational.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics.Contracts;

namespace MathLibrary
{
    public class Rational
    {
        int num;
        int den;

        public Rational(int numerator, int denominator)
        {

            Contract.Requires(0 < denominator);

            Contract.Requires(0 <= numerator);

            numerator = -1;

            this.num = numerator;

            this.den = denominator;

        }

        [ContractInvariantMethod]

        private void ObjectInvariant()
        {
            Contract.Invariant(this.den > 0);
            Contract.Invariant(this.num >= 0);
        }

        public int ToInt()
        {
            Contract.Ensures(Contract.Result<int>() >= 0);
            return this.num / this.den;
        }
    }

    class Program
    {

        static void Main(string[] args)
        {
            Rational rational = new Rational(12, 3);
        }

    }

}
```

Search.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace MathLibrary
{
        public class Search
        {
            public static int BinarySearch(int[] array, int value)
            {
                int inf = 0;
                int sup = array.Length - 1;

                while (inf <= sup)
                {
                    int index = (inf + sup) >> 1;

                    int mid = array[index];

                    if (value == mid)
                        return index;
                    if (mid < value)
                        inf = index + 1;
                    else
                        sup = index - 1;
                }
                return -1;
            }
        }
}
```

TriangleClassification.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics.Contracts;

namespace MathLibrary
{

    public class TriangClassification
    {
        public TriangClassification()
        {
        }

        int triang(int i, int j, int k)
        {
            if ((i <= 0) || (j <= 0) || (k <= 0))
            {
                return 4;
            }

            int tri = 0;

            if (i == j)
            {
                tri += 1;
            }

            if (i == k)
            {
                tri += 2;
            }

            if (j == k)
            {
                tri += 3;
            }

            if (tri == 0)
            {
                if ((i + j == k) || (j + k <= i) || (i + k <= j))
                {
                    tri = 4;
                }
                else
                {
                    tri = 1;
                }
            }
            else
            {
                if (tri > 3)
                {
                    tri = 3;
```

```
            }
            else
            {
                if ((tri == 1) && (i + j > k))
                {
                    tri = 2;
                }
                else
                {
                    if ((tri == 2) && (i + k > j))
                    {
                        tri = 2;
                    }
                    else
                    {
                        if ((tri == 3) && (j + k > i))
                        {
                            tri = 2;
                        }
                        else
                        {
                            tri = 4;
                        }
                    }
                }
            }
        }

        return tri;
    }
}

public class FindMax
{
    public int getMax(int num1, int num2, int num3, int num4)
    {
        int max = 0;
        if (num1 > num2)
        {
            max = num1;
        }
        else
        {
            max = num2;
        }

        if (max < num3)
        {
            max = num3;
            if (max < num4)
            {
                max = num3;
            }
        }
        else
        {
            if (max < num4)
            {
                max = num4;
            }
        }
```

```
            return max;
        }
    }
}
```

CompareParadigm.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Diagnostics.Contracts;

namespace MathLibrary
{
    class CompareParadigm
    {
        int num;
        int den;

        public CompareParadigm(int numerator, int denominator)
        {
            Contract.Requires(0 < denominator);
            Contract.Requires(0 <= numerator);
            Contract.Requires(numerator > denominator);
            this.num += numerator;
            this.den = denominator;
        }


        [ContractInvariantMethod]
        private void ObjectInvariant()
        {
            Contract.Invariant(this.den > 0);
            Contract.Invariant(this.num >= 0);
        }

        // Method returns the closest integer by truncation

        public int ToInt()
        {

            Contract.Ensures(Contract.Result<int>() >= 0);

            return this.num * this.den;

        }

    }

    class Run
    {
        static void Main(string[] args)
        {
            int num1 = 0;
            int num2 = 0;

            num2 = 3;
            num1 = num2 + 2;

            CompareParadigm rational = new CompareParadigm(num1, num2);
```

```
            }
        }
    }
```