Ατομική Διπλωματική Εργασία

ΑΥΤΟΜΑΤΟΠΟΙΗΣΗ ΕΛΕΓΧΟΥ ΛΟΓΙΣΜΙΚΟΥ ΜΕ ΤΗΝ ΧΡΗΣΗ ΓΡΑΦΟΥ ΡΟΗΣ ΔΕΔΟΜΕΝΩΝ ΚΑΙ ΓΕΝΕΤΙΚΩΝ ΑΛΓΟΡΙΘΜΩΝ

ΚΡΟΚΟΥ ΑΝΤΡΙΑ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2008

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Αυτοματοποίηση ελέγχου λογισμικού με την χρήση γράφου ροής δεδομένων και γενετικών αλγορίθμων

Κρόκου Άντρια

Επιβλέπων Καθηγητής
Αντρέας Αντρέου

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Μάιος 2008

Ευχαριστίες

Για την επιτυχή ολοκλήρωση της διπλωματικής αυτής έχουν συνεισφέρει κάποια άτομα το οποία νιώθω υποχρέωση μου να τα ευχαριστήσω. Καταρχάς θα ήθελα να εκφράσω βαθύτατες ευχαριστίες στον επιβλέποντα καθηγητή μου, κύριο Αντρέα Αντρέου ο οποίος με καθοδηγούσε τόσο καθ' όλη την διάρκεια της ατομικής διπλωματικής μου όσο και καθ' όλη την διάρκεια φοίτησης μου στο Πανεπιστήμιο Κύπρου. Οι συμβουλές και κατευθυντήριες γραμμές που μου υποδείκνυε έπαιξαν καθοριστικό ρόλο για την επιτυχή ολοκλήρωση των σπουδών μου.

Επίσης πολλές βαθύτατες ευχαριστίες θα ήθελα να εκφράσω στον κύριο Αναστάση Σοφοκλέους με τον οποίο βρισκόμουν σε συνεχή επαφή και συνεννόηση. Θέλω επίσης να τον ευχαριστήσω για τις ώρες που μου αφιέρωσε για να αντιμετωπίσουμε τα διάφορα προβλήματα που μου παρουσιάζονταν συνεχώς. Η βοήθεια του ήταν μεγάλης σημασίας για μένα.

Επίσης ευχαριστώ πάρα πολύ τον Κύπρο Οικονομίδη ο οποίος με κατατόπισε στο αρχικό στάδιο της διπλωματικής μου και με βοήθησε πολύ να καταλάβω τι έκανε αυτός στο μεταπτυχιακό του ούτως ώστε να μπορώ να το συνεχίσω κι εγώ με την διπλωματική μου.

Όλη η πορεία μου μέχρι την ολοκλήρωση των σπουδών μου ήταν δύσκολη και επίπονη. Θέλω να ευχαριστήσω την οικογένεια μου και του φίλους μου που με στήριξαν μέχρι τέλος.

Τέλος ευχαριστώ τον Θεό που με αξίωσε να ολοκληρώσω τις σπουδές μου και που με όπλισε με υπομονή και επιμονή για να αντιμετωπίσω τις δυσκολίες που μου παρουσιάζονταν.

Περίληψη

Αυτή η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου.

Σκοπός της εργασίας αυτής είναι η αυτοματοποίηση ελέγχου λογισμικού. Όπως όλοι γνωρίζουμε ένα λογισμικό αναπόφευκτα προκύπτει από την ανάπτυξή του με ανεπιθύμητα ελαττώματα. Αυτό που όμως είναι δυνατό να επιτευχθεί, είναι η μείωση της πιθανότητας αποτυχίας του έργου λογισμικού, η οποία θα επιφέρει και ελάττωση της αβεβαιότητας που ενυπάρχει σε αυτό.

Η εργασία αυτή έχει ενσωματωθεί και επεκτείνει ένα υπάρχον σύστημα αυτοματοποίησης ελέγχου το οποίο είχε αρχικά αναπτυχθεί από τον Σοφοκλέους Αναστάση και στην συνέχεια από τον Οικονομίδη Κύπρο.

Στηριζόμενη στα αποτελέσματα του συστήματος αυτού έχω αναπτύξει ένα κριτήριο ελέγχου που αναφέρεται στην διεθνή βιβλιογραφία για έλεγχο λογισμικού χρησιμοποιώντας τον γράφο ροής δεδομένων που ήταν ήδη υλοποιημένος. Πιο συγκεκριμένα στην εργασία αυτή έχει υλοποιηθεί το κριτήριο του Ntafos, το οποίο είναι κριτήριο παραγωγής μονοπατιών μέσω γράφων ροής δεδομένων. Στην συνέχεια, με την βοήθεια γενετικών αλγορίθμων παράγονται δεδομένα εισόδου (test cases) με στόχο την κάλυψη των μονοπατιών που παράχθηκαν από το κριτήριο του Ntafos, με απώτερο σκοπό την εκτίμηση της ποιότητας ενός προγράμματος. Όσο πιο ψηλή είναι η κάλυψη τόσο πιο ψηλή είναι και η ποιότητα του λογισμικού. Αν η κάλυψη είναι χαμηλή τότε ίσως υπάρχει dead code στον κώδικα κ.λ.π. Η αρχιτεκτονική που χρησιμοποιήθηκε και γενικά η διεπαφή χρήστη (interface), είναι βασισμένη σε αυτή που δημιουργήθηκε στα πλαίσια άλλων Ατομικών Διπλωματικών Εργασιών, με ελάχιστες αλλαγές.

Περιεχόμενα

Κεφάλαιο 1 1
Εισαγωγή 1
1.1. Γενικά 1
1.1.1. Τι είναι Μηχανική-Τεχνολογία λογισμικού 1
1.1.2. Τι είναι λογισμικό 2
1.1.4. Κύκλος ζωής λογισμικού 3
1.1.5. Γιατί χρειάζεται ο έλεγχος λογισμικού 7
1.2. Υποκίνηση-Ώθηση εργασίας 9
1.3. Σκιαγράφηση εργασίας 10
Κεφάλαιο 2 12
Έλεγχος λογισμικού 12
2.1. Εισαγωγικά 12
2.2. Σύντομη ιστορική αναδρομή ελέγχου 13
2.3. Ανάλυση ρίσκου 13
2.4. Τεχνικές ελέγχου 14
2.4.1. Έλεγχος Μαύρου Κουτιού - Black Box Testing 15
2.4.2. Έλεγχος Άσπρου Κουτιού – White Box Testing 18
2.4.3. Έλεγχος Γκρίζο κουτί - Gray Box Testing 20
2.4.4. Πολυπλοκότητα κώδικα - Cyclomatic Complexity 21
2.4.5. Στατικός έλεγχος – Statistic Testing 22
2.4.6. Δυναμικός Έλεγχος - Dynamic Testing 23
2.5. Επίπεδα και φάσεις του ελέγχου λογισμικού 23
2.5.1. Έλεγχος μονάδων – Unit Testing 23
2.5.2. Έλεγχος ολοκλήρωσης – Integration testing 24
2.5.3 Λειτουργικός Έλεγχος ή Έλεγχος Συστήματος – Functional/System Testing 24
2.5.4. Έλεγχος αποδοχής – Acceptance Testing 25
2.5.5. Επανέλεγχος – Regression Testing 25
2.5.6. Beta Testing 26
2.6. Υλοποίηση ελέγχου 27
2.7. Πώς ξέρουμε ότι τέλειωσε ο έλεγχος 28
2.8. Μέτρηση αποτελεσματικότητας ελέγχου 29
2.8.1. Ικανοποίηση πελάτη 29
2.8.2. Ατέλειες που βρέθηκαν στον έλεγχο 29
2.8.3. Κάλυψη λογισμικού 30

2.9. Κριτήρια Επάρκειας Ελέγχου (Test adequacy criteria)....31
2.9.1. Κριτήρια ελέγχου....32
2.10. Κριτήρια κάλυψης ροής ελέγχου (Control flow coverage)....34
2.10.1. Κάλυψη εντολής (Statement Coverage)....34
2.10.2. Κάλυψη Ακμής (Edge Coverage)....35
2.10.3. Κάλυψη Συνθήκης (Condition Coverage)....35
2.10.4. Κάλυψη Ακμής/ Συνθήκης (Edge/Condition Coverage)....36
2.10.5. Κάλυψη Πολλαπλής Συνθήκης (Multiple Condition Coverage)....36
2.10.6. Κριτήριο Κάλυψης Μονοπατιού (Path Coverage Criteria)....36
2.10.7. Μειονεκτήματα κριτηρίων κάλυψης....37
2.10.8. Σύγκριση των κριτηρίων κάλυψης....37
2.11. Τι είναι οι γράφοι ροής δεδομένων....38
2.12. Συνθετικότητα (Compositionality)....39
2.12.1. Έλεγχος Bottom-up....39
2.12.2. Έλεγχος Top-down....39
Κεφάλαιο 3....41
Γράφος ροής Δεδομένων....41
3.1. Εισαγωγικά....41
3.2. Βασικές έννοιες....42
3.2.1. Ορισμός του γράφου ροής δεδομένων....42
3.2.2. Τι είναι στατική ανάλυση....42
3.2.3. Ορισμός και χρήση μεταβλητής....43
3.2.4. Υποθέσεις και παραδοχές....47
3.3. Σημαντικότερες μέθοδοι πλοήγησης σε γράφο....48
3.3.1. Αναζήτηση σε βάθος (Depth first search)....48
3.3.2. Αναζήτηση σε πλάτος (Breadth first search)....49
3.4. Το κριτήριο του Ntafos....50
3.4.1. Επιπρόσθετοι ορισμοί....50
3.4.2. Ορισμός κριτηρίου Ntafos....50
3.4.3. Πως ξέρω ποιο είναι το μέγιστο k-dr interaction....53
3.4.4. Παράδειγμα κριτηρίου Ntafos....54
3.5. Τα κριτήρια των Rapps και Weyuker....56
3.5.1. Μερικά παραδείγματα κάποιων κριτηρίων....61
3.6. Τα κριτήρια Laski και Korel....63
3.6.1. Επιπρόσθετοι ορισμοί....63
3.6.2. Τα κριτήρια των Laski και Korel....66
3.7. Σύγκριση των κριτηρίων....67

Κεφάλαιο 4 69
Γενετικοί αλγόριθμοι 69
4.1. Γενικά 69
4.2. Ιστορική αναδρομή γενετικών αλγορίθμων 70
4.3. Βιολογική ορολογία 71
4.4. Πως λειτουργούν 73
4.5. Περιγραφή τρόπου υλοποίησης γενετικών αλγορίθμων 73
4.5.1. Δημιουργία Αρχικού Πληθυσμού 74
4.5.2. Αξιολόγηση – Fitness 75
4.5.3. Επιλογή – Selection 75
4.5.4. Αναπαραγωγή – Reproduction 78
4.5.5. Διασταύρωση – Crossover 78
4.5.6. Μετάλλαξη - Mutation 80
4.6. Λόγοι που οδήγησαν στους γενετικούς αλγορίθμους 80
4.6.1. Πλεονεκτήματα γενετικών αλγορίθμων 80
4.7. JGAP: Βιβλιοθήκη υλοποίησης γενετικών αλγορίθμων 83
4.7.1. Τι είναι το JGAP 83
Κεφάλαιο 5 85
Παρουσίαση προηγούμενης εργασίας 85
5.1. Εισαγωγή 85
5.2. Περιγραφή προηγούμενης εργασίας 85
5.2.1. Αρχιτεκτονική αναλυτή προγραμμάτων 85
5.2.2. Παραγωγή Γράφου Ροής Δεδομένων 88
5.3. Παρόμοιες/παλιές εργασίες – Related work 90
Κεφάλαιο 6 93
Αυτοματοποίηση ελέγχου 93
6.1. Εισαγωγικά 93
6.2. Περιγραφή υλοποίησης κριτηρίου Ntafos και ψευδοκώδικας 93
6.3. Περιγραφή υλοποίησης γενετικών αλγορίθμων 102
6.3.1. Γονίδιο (gene) 102
6.3.2. Χρωμόσωμα (Chromosome) 103
6.3.3. Γονότυπος (Genotype) 103
6.3.4. Συνάρτηση Ποιότητας (Fitness function) 105
Κεφάλαιο 7 109
Εγχειρίδιο χρήσης (Manual) 109
7.1. Εισαγωγή 109
7.2. Περιγραφή οθόνων 109

Κεφάλαιο 8 120
8.1. Εισαγωγή 120
8.2. Αποτελέσματα: Ποσοστό κάλυψης μονοπατιών του κριτηρίου Ntafos 120
8.3. Σύγκριση με προηγούμενη δουλειά 129
8.4. Αποτελέσματα και συμπεράσματα 130
8.5. Μελλοντική εργασία 134
Βιβλιογραφία 1

Κεφάλαιο 1

Εισαγωγή

1.1. Γενικά 1
1.2. Υποκίνηση-Ώθηση εργασίας 9
1.3. Σκιαγράφηση εργασίας.....10

1.1. Γενικά

1.1.1. Τι είναι Μηχανική-Τεχνολογία λογισμικού

Στην σημερινή κοινωνία της πληροφορίας το κόστος ανάπτυξης λογισμικού ανεβαίνει ενώ το κόστος του hardware μειώνεται. Οι χρόνοι ανάπτυξης εφαρμογών αυξάνονται και το κόστος συντήρησης των εφαρμογών μεγαλώνει. Τα λάθη των λογισμικών αυξάνονται ενώ τα λάθη του hardware έχουν μειωθεί στο ελάχιστο. Για να μπορέσουμε να λύσουμε τα παραπάνω προβλήματα τα οποία γίνονται όλο και ποιο έντονα θα πρέπει να ακολουθούμε δομημένους τρόπους ανάπτυξης λογισμικού. Η Τεχνολογία Λογισμικού ή αλλιώς "Software Engineering" συμπεριλαμβάνει όλο το φάσμα εργασιών οι οποίες πρέπει να γίνουν για την σωστή υλοποίηση λογισμικού.

Πιο συγκεκριμένα η Μηχανική (ή Τεχνολογία) Λογισμικού είναι ένας νέος κλάδος της επιστήμης των Η/Υ που όμως πραγματεύεται προβλήματα με τεράστια πολυπλοκότητα. Ιστορικά ο όρος Μηχανική Λογισμικού κάνει την εμφάνιση του στη βιβλιογραφία το 1968 και αποδίδεται στον Fritz Bauer. Αναφέρθηκε για πρώτη φορά σ' ένα συνέδριο στο Garmisch. Στο συνέδριο, το κύριο θέμα συζήτησης ήταν η «κρίση του λογισμικού» ("the software crisis"). Ο Bauer όρισε την μηχανική λογισμικού ως εξής:

> *«Η καθιέρωση και η χρήση υγιών αρχών εφαρμοσμένης μηχανικής προκειμένου να παραχθεί οικονομικά, λογισμικό που είναι αξιόπιστο και λειτουργεί αποτελεσματικά σε πραγματικές μηχανές» [3]*

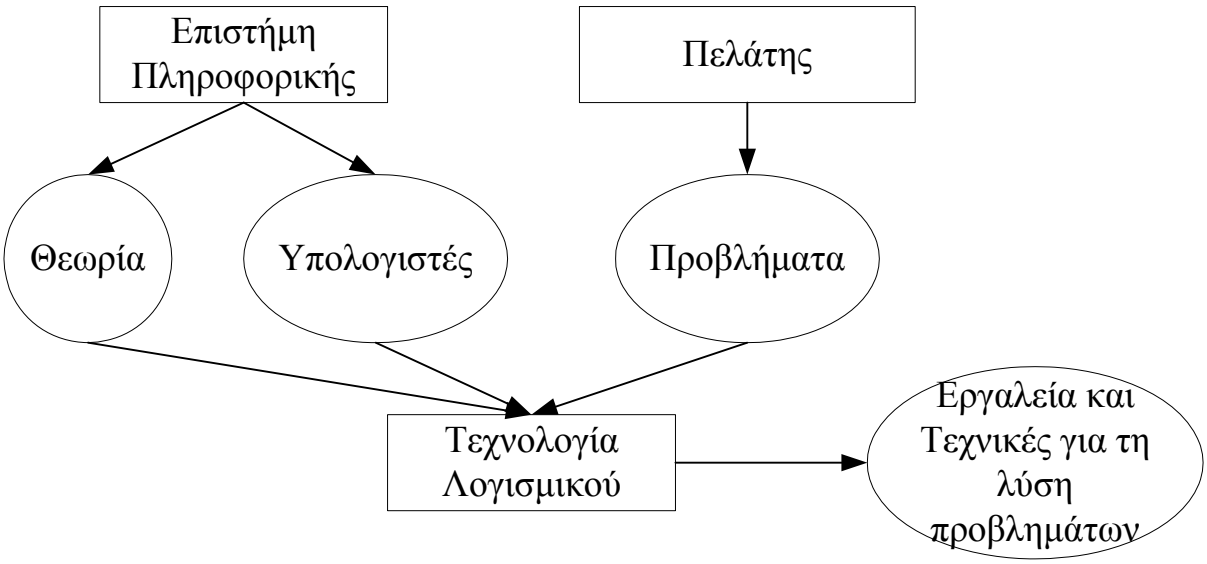
Στο [2] αναφέρεται ότι η Μηχανική (ή Τεχνολογία) Λογισμικού (Software Engineering) είναι η εφαρμογή μιας συστηματικής, πειθαρχημένης, ποσοτικά προσδιορισμένης προσέγγισης στην ανάπτυξη, λειτουργία, και συντήρηση του λογισμικού. Οι αρχές της μηχανικής λογισμικού περιλαμβάνουν τη γνώση, τα εργαλεία,

και τις απαραίτητες μεθόδους για την ανάλυση απαιτήσεων του λογισμικού, το σχεδιασμό και την κατασκευή του, τη δοκιμή και επαλήθευση του, και τέλος την συντήρηση του λογισμικού. Άλλοι ορισμοί για την μηχανική λογισμικού είναι:

Ο Ian Somerville στο [1] αναφέρει:

> *«Μηχανική Λογισμικού είναι μια ειδικότητα εφαρμοσμένης μηχανικής που ενδιαφέρεται για όλες τις πτυχές της παραγωγής λογισμικού από τα αρχικά στάδια της προδιαγραφής συστημάτων στη διατήρηση του συστήματος αφότου έχει περάσει σε χρήση».*

Πιο κάτω παρουσιάζω (πέρα από τους ορισμούς) ένα τυπικό διάγραμμα που αντιπροσωπεύει τι είναι τεχνολογία λογισμικού:



Σχήμα 1.1: Τι είναι τεχνολογία λογισμικού

1.1.2. Τι είναι λογισμικό

Το λογισμικό είναι ένα σύνολο εντολών προς τον υπολογιστή που όταν εκτελεστούν παρέχουν επιθυμητές λειτουργίες και αποδόσεις. Το λογισμικό επίσης εκτός από διάφορες εντολές μπορεί να αποτελείται και από διάφορες δομές δεδομένων που επιτρέπουν την ικανοποιητική διαχείριση πληροφοριών. Συνήθως διάφορα έγγραφα συνοδεύουν την παράδοση του λογισμικού προς τον πελάτη, τα οποία έγγραφα περιγράφουν την λειτουργία και χρήση των διάφορων προγραμμάτων.

1.1.3. Τι είναι υπολογιστική νοημοσύνη

Η υπολογιστική νοημοσύνη είναι ο διάδοχος της Τεχνητής νοημοσύνης. Η υπολογιστική νοημοσύνη συνδυάζει στοιχεία μάθησης (learning), προσαρμογής (adaptation), εξέλιξης (evolution) και ασαφούς λογικής (fuzzy logic) για να δημιουργήσει προγράμματα που είναι κατά κάποιο τρόπο «έξυπνα». Η παρούσα εργασία, η οποία εμπλέκει υπολογιστική νοημοσύνη, συνδυάζεται με τους όρους εξέλιξη και προσαρμογή οι οποίοι παρέχονται μέσω των γενετικών αλγορίθμων (περισσότερα παρουσιάζονται στο κεφάλαιο 4).

1.1.4. Κύκλος ζωής λογισμικού

Ένα πρόγραμμα αρχίζει την ζωή του από την στιγμή που θα καθοριστούν οι απαιτήσεις του, οι προδιαγραφές του και παύει να ζει όταν εξαντληθούν όλα τα περιθώρια συντήρησής του (προσθήκες, αλλαγές, βελτιώσεις).

Θα έχετε προσέξει ότι στα διάφορα πακέτα λογισμικού, είτε είναι γλώσσες προγραμματισμού, είτε πακέτα εφαρμογών, είτε λειτουργικά συστήματα, δίπλα στο εμπορικό όνομα του λογισμικού, υπάρχει ο αριθμός της έκδοσής του (version).

Ο αριθμός έκδοσης που συνοδεύει την ονομασία κάθε πακέτου λογισμικού, δείχνει ακριβώς τις αλλαγές που έχουν πραγματοποιηθεί από την αρχική του εμφάνιση. Ο κανόνας λέει ότι, όταν οι αλλαγές είναι σημαντικές, δηλαδή έχουν προστεθεί νέες λειτουργίες, εντολές, προγράμματα, ο αριθμός αυξάνει κατά ακέραιο αριθμό (π.χ. DOS ver.5 DOS ver.6), όταν οι αλλαγές είναι μικρότερες, τότε αυξάνεται κατά δέκατα (Windows v.3.1, Windows v.3.11). Τα τελευταία χρόνια αυτή η εξέλιξη των δυνατοτήτων ενός λογισμικού έχει συνδεθεί με την πολιτική πωλήσεων των εταιρειών παραγωγής. Έτσι οι αριθμοί που ακολουθούν τις ονομασίες του λογισμικού, έχουν έντονη την χροιά του τμήματος πωλήσεων και όχι του τμήματος ανάπτυξης της εταιρείας (Windows 95, Windows98, Office 97 κ.λπ).

Θα έχετε παρατηρήσει ακόμα ή θα έχετε ακούσει, ότι το τάδε λογισμικό αντικαταστάθηκε από το δείνα, για παράδειγμα το MS-DOS αντικαταστάθηκε από τα Windows.

Οι αριθμοί έκδοσης και οι αντικαταστάσεις του λογισμικού δείχνουν με τον καλύτερο τρόπο ότι, κάθε λογισμικό έχει ένα κύκλο ζωής, όπου στον κύκλο αυτό γεννιέται και πεθαίνει όπως ένας ζωντανός οργανισμός.

Τι σημαίνει όμως "κύκλος ζωής"; Ποιες είναι οι ενδιάμεσες φάσεις του; Μια σύγκριση θα μας δώσει καλύτερα το περιεχόμενο του όρου "Κύκλος Ζωής". Ας δούμε πρώτα τον "κύκλο ζωής" ενός αυτοκινήτου.

Η δημιουργία ενός αυτοκινήτου ξεκινά συνήθως από μία έρευνα που θα καταγράψει τις επιθυμίες του κοινού, αυτοκίνητο πόλης, εκτός δρόμου, φθηνό,

γρήγορο και την τάση της αγοράς. Τα αποτελέσματα της έρευνας, σε συνδυασμό με άλλα δεδομένα, όπως άλλα μοντέλα του εργοστασίου, μοντέλα του ανταγωνισμού κ.λ.π., θα καθορίσουν τις προδιαγραφές του νέου αυτοκινήτου, μικρό ή μεγάλο, γρήγορο, νεανικό, με πόσες πόρτες, με ποια πρόσθετα, σε ποια τιμή κ.λπ. (**φάση ανάλυσης**).

Οι μηχανικοί της εταιρείας έχοντας υπ' όψη τις προδιαγραφές θα σχεδιάσουν το νέο αυτοκίνητο επιλέγοντας τα κατάλληλα υλικά, δίνοντάς του την εξωτερική μορφή και τα άλλα χαρακτηριστικά που έχουν προδιαγραφεί, ιπποδύναμη, κατανάλωση, κ.λπ. (**φάση σχεδιασμού**).

Το τμήμα παραγωγής του εργοστασίου θα πάρει τα σχέδια και θα προσαρμόσει τη γραμμή παραγωγής από πλευράς διαδικασιών και ελέγχων, ώστε να μπορεί να παράγεται το νέο αυτοκίνητο (**φάση υλοποίησης**).

Το έτοιμο αυτοκίνητο θα πρέπει να περάσει ένα τελικό έλεγχο λειτουργίας στο δρόμο και κάτω από ακραίες συνθήκες. Αυτός ο έλεγχος θα διενεργηθεί μέσα στις εγκαταστάσεις της εταιρείας, ώστε το νέο αυτοκίνητο να δοκιμαστεί σαν σύνολο (**φάση ελέγχου**).

Το δοκιμασμένο αυτοκίνητο είναι έτοιμο για πώληση. Πωλούμενο μπαίνει πλέον σε κανονική λειτουργία, η οποία εξασφαλίζεται με την κατάλληλη συντήρηση από τους εξουσιοδοτημένους μηχανικούς (**φάση λειτουργίας και συντήρησης**).

Αυτός λοιπόν είναι ο κύκλος ζωής ενός αυτοκινήτου, παρόμοιος σε αρκετά σημεία με τον κύκλο ζωής ενός προγράμματος, ο οποίος είναι:

- Προσδιορισμός-ανάλυση απαιτήσεων λογισμικού (*software requirements*) από τον πελάτη
- Σχεδίαση (*design*)
- Κωδικοποίηση (*coding*) -Υλοποίηση
- Έλεγχος (*testing*)
- Παράδοση συστήματος (*delivery*)
- Λειτουργία (*operation*)
- Συντήρηση (*maintenance*)
- Απόσυρση (*abortion*)

Το τέλος κάθε φάσης οδηγεί στην επόμενη, αλλά μπορεί να οδηγήσει και στην προηγούμενη. Όταν το τέλος μιας φάσης οδηγεί στην προηγούμενη, σημαίνει ότι ορισμένα στοιχεία, χρειάζεται να επανακαθοριστούν.

Στη **φάση Ανάλυσης και Σχεδίασης** που ακολουθεί τον ορισμό του προβλήματος από τον πελάτη, μπορούμε να διακρίνουμε τις εξής ενέργειες:

- Καταγράφονται αναλυτικά τα δεδομένα και τα ζητούμενα του προβλήματος.
- Ζητούνται οι απαραίτητες διευκρινήσεις από τον πελάτη, σε όσα σημεία οι προδιαγραφές παρουσιάζουν ασάφεια.
- Καθορίζεται η δομή του προγράμματος.
- Καθορίζονται οι ενότητες (ρουτίνες, υποπρογράμματα) από τις οποίες θα αποτελείται το πρόγραμμα.
- Αναζητούνται έτοιμες ενότητες (modules) από παλιότερα προγράμματα που μπορούν να χρησιμοποιηθούν και σ' αυτό το πρόγραμμα.
- Επιλέγονται οι αλγόριθμοι και οι δομές δεδομένων που θα χρησιμοποιηθούν σε κάθε ενότητα.

Στην επόμενη **φάση της Υλοποίησης** του προγράμματος σε κάποια γλώσσα προγραμματισμού, ακολουθούμε τα εξής βήματα με τη σειρά:

- Επιλέγεται η γλώσσα προγραμματισμού για το συγκεκριμένο πρόγραμμα. Σημειώστε ότι όλες γλώσσες δεν είναι κατάλληλες για όλα τα προβλήματα.
- Εισάγεται το κωδικοποιημένο πρόγραμμα στον υπολογιστή. Το πρόγραμμα αυτό είναι το αρχικό πρόγραμμα (source program).
- Ζητείται η μετάφραση του προγράμματος από ένα μεταγλωττιστή, ώστε αυτό να γίνει κατανοητό από τον υπολογιστή. Το πρόγραμμα αυτό είναι το τελικό πρόγραμμα (object program).
- Η μετάφραση θα αποκαλύψει λάθη "ορθογραφίας" και "συντακτικού" της γλώσσας προγραμματισμού.
- Διορθώνονται τα λάθη και ακολουθεί ξανά μετάφραση του προγράμματος, έως την οριστική εξάλειψή τους.

Προσοχή: Ο έλεγχος ενός προγράμματος, διαπιστώνει την ύπαρξη ενός λάθους, ποτέ όμως δεν πιστοποιεί την ανυπαρξία λάθους.

Η φάση των **Ελέγχων** αναφέρεται στον έλεγχο των λογικών λαθών που πιθανόν να υπάρχουν σε ένα πρόγραμμα. Η σειρά των ελέγχων είναι η ακόλουθη.

- Το πρόγραμμα ελέγχεται, με τα δεδομένα ελέγχου που είχε ελεγχθεί και ο αλγόριθμος, για να διαπιστωθεί αν παράγει τα επιθυμητά αποτελέσματα.
- Διαπιστώνονται λάθη που οφείλονται είτε σε λάθη κατά την κωδικοποίηση του αλγόριθμου είτε από λανθασμένη επικοινωνία ενοτήτων.
- Τα τυχόν λάθη διορθώνονται και οι έλεγχοι συνεχίζονται μέχρι το πρόγραμμα να απαλλαγεί από αυτά.

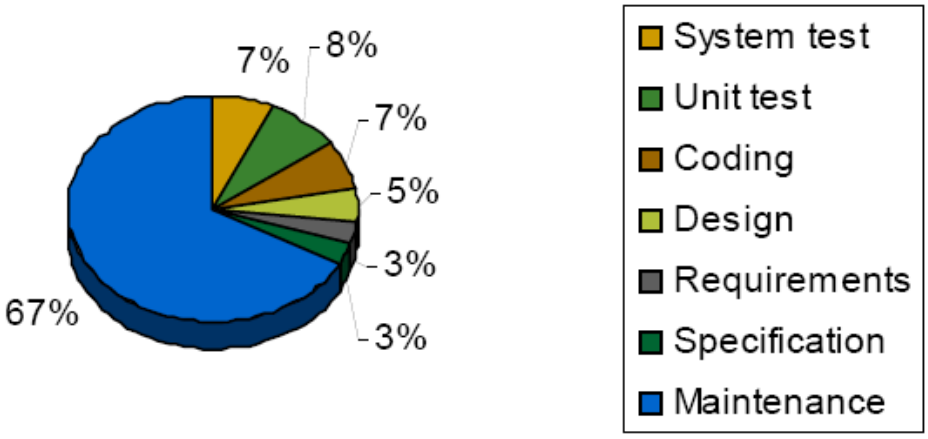
- Παρουσιάζονται τα αποτελέσματα στον πελάτη προκειμένου να έχουμε την οριστική επικύρωση εκ μέρους του.
- Αν παρουσιάζεται απόκλιση από τις προδιαγραφές, θα πρέπει να επιστρέψουμε στην φάση της ανάλυσης και σχεδίασης προκειμένου να γίνουν οι κατάλληλες διορθώσεις.

Στο τέλος της φάσης των ελέγχων έχουμε ένα έτοιμο πρόγραμμα που μπορούμε να παραδώσουμε στον πελάτη (**φάση παράδοσης συστήματος**) για χρήση. Δεν πρέπει να ξεχνάμε, ότι ο έλεγχος του προγράμματος διαπιστώνει την ύπαρξη ενός λάθους που πρέπει να διορθωθεί, δυστυχώς όμως δεν πιστοποιεί την απουσία λάθους. Έτσι ένα πρόγραμμα κατ' ουσία βρίσκεται κάτω από ένα διαρκή έλεγχο της ορθής λειτουργίας του.

Στην επόμενη φάση της **Συντήρησης** γίνονται όλες οι προσαρμογές και βελτιώσεις που χρειάζονται προκειμένου το πρόγραμμα να συνεχίσει να χρησιμοποιείται. Η φάση αυτή διαρκεί όσο θα χρησιμοποιείται το πρόγραμμα και είναι η φάση που παίρνει και τον περισσότερο χρόνο (Σχήμα 1.2). Εδώ θα παρατηρήσουμε ότι:

- Οι προσαρμογές είναι αναπόφευκτες όταν διαφοροποιούνται τα δεδομένα του προβλήματος ή όταν ο χρήστης ζητήσει νέες λειτουργίες.
- Κάποιες προσαρμογές μπορεί να απαιτήσουν την εκτέλεση της φάσης της ανάλυσης και σχεδίασης και άρα όλων των υπολοίπων φάσεων.
- Οι βελτιώσεις προκύπτουν από την εμπειρία που αποκτάται με τον καιρό και μας κάνει να "βλέπουμε" τα ίδια πράγματα με "άλλο μάτι".
- Κάθε προσαρμογή ή βελτίωση θα πρέπει να καταλήγει σε συνολικό έλεγχο του προγράμματος και φυσικά στην καταγραφή των σχετικών σχολίων για την τεκμηρίωση.
- Το τελευταίο στάδιο αυτής της φάσης έρχεται με την τελειοποίηση του προγράμματος. Σ' αυτή τη φάση το πρόγραμμά δουλεύει υποδειγματικά … μέχρι την επόμενη αλλαγή.

Και τελευταία φάση του κύκλου ζωής λογισμικού είναι η φάση της **απόσυρσης** όπου το λογισμικό αποσύρεται και παύει να χρησιμοποιείται [27].



Σχήμα 1.2: Κατανομή χρόνου σε ένα τυπικό έργο

1.1.5. Γιατί χρειάζεται ο έλεγχος λογισμικού

Το λογισμικό τώρα, το οποίο είναι ένα πολύπλοκο πνευματικό προϊόν, αναπόφευκτα προκύπτει από την ανάπτυξή του με ανεπιθύμητα ελαττώματα (defects), τα οποία είναι πιθανό να οφείλονται σε ενδογενείς και εξωγενείς αιτίες και των οποίων η μη ανακάλυψη και κατάλληλη αντιμετώπιση μπορεί να προκαλέσει συνέπειες που μπορούν να ποικίλουν από απλά ενοχλητικές έως και καταστροφικές. Τεράστιες ποσότητες πόρων ξοδεύονται σε όλες τις σχετικές με τη τεχνολογία λογισμικού βιομηχανίες με σκοπό την αποφυγή αποτυχιών λογισμικού. Παρόλα αυτά, τελικά είναι αδύνατο να υπάρξει η εγγύηση ότι το λογισμικό είναι τέλειο, όπως είναι επίσης αδύνατο να προβλεφθεί και να εξαλειφθεί κάθε τι από τον εξωτερικό κόσμο που πιθανόν να απειλήσει το λογισμικό όσο αυτό εκτελείται.

Αυτό που όμως είναι δυνατό να επιτευχθεί, είναι η μείωση της πιθανότητας αποτυχίας του έργου λογισμικού, η οποία θα επιφέρει και ελάττωση της αβεβαιότητας που ενυπάρχει σε αυτό. Προϋπόθεση για την επίτευξη αυτής της ελάττωσης αποτελεί η εφαρμογή ενός κατάλληλου ελέγχου καθόλη τη διάρκεια της ανάπτυξης του λογισμικού αλλά και ενός κατάλληλου ελέγχου μετά την ολοκλήρωση του λογισμικού ώστε να επιτευχθεί επαρκής αναγνώριση και αποτελεσματική αντιμετώπιση των διαφόρων κινδύνων που απειλούν το σύστημα λογισμικού.

Σύμφωνα με τα πιο πάνω προκύπτει ότι ο έλεγχος του λογισμικού αποτελεί αναπόσπαστο κομμάτι της διαδικασίας παραγωγής λογισμικού. Δεν είναι λίγες οι φορές που λόγω έλλειψης χρόνου ή λανθασμένης αντίληψης για την σημασία του ελέγχου λογισμικού, το κομμάτι αυτό παραμελείτε ή παραβλέπεται. Λογισμικό χρησιμοποιείται σε κάθε πτυχή της καθημερινής μας ζωής: στην οικονομία, στην εκπαίδευση, στην ιατρική, στο περιβάλλον. Λόγω αυτής της ραγδαίας και ευρείας εξάπλωσής τους, ο σωστός έλεγχος λογισμικού είναι πολύ σημαντικός ειδικά για λογισμικά που ασχολούνται με θέματα «ζωτικής σημασίας».

Η διαδικασία ελέγχου λογισμικού είναι πολύ σημαντική αλλά ταυτόχρονα είναι μια δύσκολη, επίπονη και δαπανηρή εργασία. Περίπου 30-50% του συνολικού κόστους στην διαδικασία διεκπεραίωσης ενός λογισμικού προγράμματος, αφιερώνεται στο έλεγχο, ενώ ο χρόνος και η προσπάθεια που ξοδεύεται στον έλεγχο ενός συστήματος μπορεί να μεγαλύτερος από αυτόν της κατασκευής του. Το κόστος αυτό αυξάνεται όταν ο έλεγχος γίνεται με τον παραδοσιακό τρόπο ελέγχου, δηλαδή χειρωνακτικά. Ταυτόχρονα, με τη χρήση αυτής της τεχνικής, η διαδικασία ελέγχου γίνεται πιο χρονοβόρα και επιρρεπής σε λάθη, με επακόλουθο να μην αποδίδει έλεγχο υψηλής ακρίβειας και ποιότητας. Ο έλεγχος με αυτό το τρόπο όσο αναλυτικός και μεγάλος σε διάρκεια και αν είναι, δεν μπορεί να δείξει ποτέ την έλλειψη λαθών, μόνο την παρουσία τους!

Από την σημασία της ορθότητας λογισμικού, και τις δυσκολίες που υπάρχουν στον «χειρωνακτικό» έλεγχο, προκύπτει ότι πρέπει να βρεθεί και να αναπτυχθεί νέος τρόπος ελέγχου λογισμικού πιο αποδοτικός και που να μπορεί να ανακαλύψει πιθανά λάθη σε πηγαίο κώδικα πιο γρήγορα. Αν η διαδικασία ελέγχου μπορούσε να γίνεται αυτοματοποιημένα, τότε όχι μόνο το κόστος ανάπτυξης λογισμικού θα μειωνόταν σημαντικά αλλά και η ποιότητα του ελέγχου και επομένως του τελικού προϊόντος θα ήταν υψηλότερη. Ένα αυτοματοποιημένο σύστημα ελέγχου πηγαίου κώδικα πρέπει να είναι σε θέση να παράγει δεδομένα εισόδου, στοιχεία δοκιμής για το υπό εξέταση σύστημα, αυτόματα με διάφορες μεθόδους. Μέχρι σήμερα έχουν παρουσιαστεί πολλές τεχνικές παραγωγής δεδομένων εισόδου.

Έχοντας αποδεχθεί το γεγονός ανάγκης ελέγχου και εφόσον ο έλεγχος αποτελεί ούτως ή άλλως μια καθοριστική φάση της ανάπτυξης του λογισμικού, η παρούσα εργασία, επιχειρεί να εστιαστεί στη φάση του ελέγχου. Κατά τον έλεγχο κρίνεται κατά πόσο μια σειρά από δεδομένα εισόδου εξετάζουν και ελέγχουν ένα κομμάτι κώδικα. Στόχος είναι να ανακαλυφθούν όσο περισσότερα λάθη γίνεται, με ένα υποψήφιο σύνολο δεδομένων εισόδου. Γι' αυτό, ένα σύνολο από δεδομένα που είναι πιθανόν να ανακαλύψει περισσότερα λάθη από κάποιο άλλο σύνολο, σημαίνει πως το πρώτο σύνολο είναι καλύτερο. Δυστυχώς είναι αδύνατον να γνωρίζει κάποιος ποσοτικά τα λάθη, που πιθανόν να ανακαλυφθούν από ένα σύνολο τιμών εισόδου. Αυτό οφείλεται όχι μόνο στην ποικιλία λαθών, αλλά και στην απροσδιόριστη σημασία του λάθους. Αυτό οδήγησε στην δημιουργία κριτηρίων κάλυψης ελέγχου, με τα οποία μπορεί να κριθεί η ποιότητα λογισμικού. Τέτοια κριτήρια μπορούν να εφαρμοστούν όταν ο υπό εξέταση κώδικας αναπαρίσταται είτε σαν γράφος ροής ελέγχου (control flow graph) είτε σαν γράφος ροής δεδομένων (data flow graph). Με τα κριτήρια κάλυψης μπορεί να ξεχωρίσει ένα καλό σύνολο τιμών εισόδου από ένα άλλο.

Η αυτοματοποίηση ελέγχου και η κάλυψη κριτηρίων όταν το πρόγραμμα αναπαρίσταται σαν γράφος ροής ελέγχου (control flow graph) και σαν γράφος ροής δεδομένων (data flow graph) έχει καλυφθεί και από προηγούμενες εργασίες [4][5]. Στόχος αυτής της εργασίας είναι να ελέγξει πως μπορούμε να εφαρμόσουμε σε γράφο ροής δεδομένων (data flow graph) μια παρόμοια μέθοδο, αξιοποιώντας κάποια κριτήρια κάλυψης που έχουν καθοριστεί σε αυτού του είδους γράφους. Πιο συγκεκριμένα σ' αυτή την εργασία θα υλοποιηθεί το κριτήριο Ntafos και θα συγκριθεί με το κριτήριο ALL-DU Paths (βλέπε πιο κάτω).

1.2. Υποκίνηση-Ώθηση εργασίας

Η αυτοματοποίηση ελέγχου είναι πολύ σημαντικό μέρος για την τεχνολογία λογισμικού για τον λόγο ότι ελαττώνει το κόστος και τον χρόνο ελέγχου λογισμικού. Έχουν δημοσιευτεί αρκετά άρθρα στα οποία έχουν προταθεί διάφορες τεχνικές ελέγχου, με την κάθε μία τεχνική να έχει τα θετικά και τα αρνητικά της.

Είναι σημαντικό να επισημανθεί ότι η πλειοψηφία των τεχνικών που έχουν προταθεί στηρίζονται σε δημιουργία γράφου ροής ελέγχου ή γράφους ροής δεδομένων και κριτήρια που εφαρμόζονται στους αντίστοιχους γράφους. Στο [5] ο Κύπρος Οικονομίδης υλοποίησε γράφο ροής δεδομένων και έδειξε πώς μπορεί να εφαρμοστεί σ' αυτόν το κριτήριο κάλυψης ALL_DU Paths το οποίο έχει οριστεί για τέτοιου είδους γράφους. Αξίζει να αναφερθεί ότι ενώ για γράφους ροής ελέγχου υπάρχουν κριτήρια κάλυψης αυστηρά ορισμένα και καθολικά αποδεκτά, κάτι τέτοιο δεν ισχύει για τους γράφους ροής δεδομένων. Για τους τελευταίους υπάρχουν διαφορετικές εναλλακτικές προτάσεις κριτηρίων κάλυψης.

Σε αυτή την ατομική διπλωματική εργασία θα υλοποιηθεί το κριτήριο Ntafos (επεξηγείται παρακάτω), το οποίο θα παράγει μονοπάτια από ένα γράφο ροής δεδομένων και με τη βοήθεια γενετικών αλγορίθμων θα παράγονται δεδομένα ελέγχου που θα καλύπτουν αυτά τα μονοπάτια. Αυτό που κάνει αυτή την εργασία να υπερτερεί από την ήδη υπάρχουσα εργασία στο [5] είναι ότι στα μονοπάτια που παράγονται με το κριτήριο του Ntafos υπάγονται και τα μονοπάτια που παράγονται με το κριτήριο ALL_DU Paths συν κάποια άλλα μονοπάτια, οπόταν μ' αυτό επιτυγχάνουμε καλύτερο έλεγχο. Αυτή η εργασία έχει επίσης ως στόχο εκτός από την υλοποίηση του κριτηρίου Ntafos και την υλοποίηση μιας καλής συνάρτησης ποιότητας (fitness function) που θα πετυχαίνει την βέλτιστη κάλυψη των μονοπατιών. Οι στόχοι της δουλειάς αυτής και οι προηγούμενες δουλειές παρουσιάζονται αναλυτικότερα στο κεφάλαιο 8.

1.3. Σκιαγράφηση εργασίας

Αυτή η εργασία ξεκινά με κάποια γενικά σχόλια για τον έλεγχο ώστε να εισάγει ομαλά τον αναγνώστη στο θέμα. Στη συνέχεια συνεχίζει με περιγραφή των γράφων ροής δεδομένων, γενετικών αλγορίθμων και κάνει μια περιγραφή της προηγούμενης δουλειάς ώστε να καταλάβει ο αναγνώστης τι γινόταν στην προηγούμενη εργασία και τι θα συνεχιστεί στην παρούσα εργασία. Ακολούθως παρουσιάζω ένα εγχειρίδιο χρήσης του εργαλείου του οποίου αναπτύχθηκε με τις αλλαγές που υπέστηκε. Στο κεφάλαιο «αυτοματοποίηση ελέγχου» παρουσιάζω την δουλειά που έγινε στην εργασία αυτή και τέλος παρουσιάζονται τα συμπεράσματα και τα αποτελέσματα. Τα κεφάλαια 2-4 λειτουργούν ως εισαγωγικά. Η δομή της μελέτης έχει την πιο ακόλουθη μορφή:

- Στο κεφάλαιο 2 γίνεται μια αναφορά για τον έλεγχο λογισμικού. Διατυπώνεται η ιστορική αναδρομή ελέγχου και παρουσιάζονται τα διάφορα είδη, τεχνικές και επίπεδα ελέγχου. Στο τέλος παρουσιάζεται σε συντομία τα κριτήρια κάλυψης των γράφων ροής ελέγχου.
- Στο κεφάλαιο 3 παρουσιάζονται τα κριτήρια κάλυψης των γράφων ροής δεδομένων και για κάποια δίνονται και κάποια παραδείγματα. Στο τέλος του κεφαλαίου αυτού ακολουθεί η σύγκριση των κριτηρίων αυτών και παρουσιάζεται η ταξινομημένη ιεραρχία τους.
- Στο κεφάλαιο 4 παρουσιάζονται οι γενετικοί αλγόριθμοι που αποτελούν σημαντικό σημείο της εργασίας αυτής. Γίνεται μια εισαγωγή σε αυτούς και παρουσιάζεται εν συντομία η κλάση η οποία χρησιμοποιήθηκε στην υλοποίηση του ολοκληρωμένου συστήματος.
- Στο κεφάλαιο 5 παρουσιάζεται η δουλειά που έγινε στην προηγούμενη εργασία. Έμφαση δίνεται στην αρχιτεκτονική του συστήματος αλλά και στην παραγωγή του γράφου ροής δεδομένων, ο οποίος παίζει σημαντικό ρόλο για την εργασία αυτή.
- Στο κεφάλαιο 6 παρουσιάζεται η διαδικασία παραγωγής των μονοπατιών βάση του κριτηρίου αλλά και το πώς γίνεται η χρήση των γενετικών αλγορίθμων. Πιο συγκεκριμένα αναλύεται πως μοντελοποιείται το γονίδιο, το χρωμόσωμα κ.λ.π. και τέλος αναλύεται και επεξηγείται η συνάρτηση ποιότητας που χρησιμοποιήθηκε στην εργασία αυτή.
- Στο κεφάλαιο 7 παρουσιάζεται ένα εγχειρίδιο χρήσης του εργαλείου που αναπτύχθηκε (για να υποστηρίζει την αυτοματοποίηση του ελέγχου) από την προηγούμενη δουλειά επεξηγώντας την χρήση και τις αλλαγές που υπέστη.
- Στο κεφάλαιο 8 το οποίο είναι και το τελευταίο κεφάλαιο της εργασίας αυτής δίνονται τα αποτελέσματα / συμπεράσματα της έρευνας αυτής και γίνεται

σύγκριση με προηγούμενη δουλειά. Τέλος γίνονται προτάσεις για μελλοντική εργασία.

Στο τέλος παρατίθεται η βιβλιογραφία και οι αναφορές προηγούμενων δουλειών.

Κεφάλαιο 2

Έλεγχος λογισμικού

2.1. Εισαγωγικά....12
2.2. Σύντομη ιστορική αναδρομή ελέγχου....13
2.3. Ανάλυση ρίσκου....13
2.4. Τεχνικές ελέγχου....14
2.5. Επίπεδα και φάσεις του ελέγχου λογισμικού....23
2.6. Υλοποίηση ελέγχου....27
2.7. Πώς ξέρουμε ότι τέλειωσε ο έλεγχος....28
2.8. Μέτρηση αποτελεσματικότητας ελέγχου....29
2.9. Κριτήρια Επάρκειας Ελέγχου (Test adequacy criteria)....31
2.10. Κριτήρια κάλυψης ροής ελέγχου (Control flow coverage)....34
2.11. Τι είναι οι γράφοι ροής δεδομένων....38
2.12. Συνθετικότητα (Compositionality)....39

2.1. Εισαγωγικά

Έλεγχος λογισμικού καλείται η διαδικασία που χρησιμοποιείται για να προσδιορίσει την ακρίβεια, την πληρότητα, την ασφάλεια και την ποιότητα ενός λογισμικού προϊόντος. Ο έλεγχος του λογισμικού είναι διαδικασία που πρέπει να εφαρμόζεται σε όλη την διάρκεια του κύκλου ζωής του, αφού μετρά και βελτιώνει την ποιότητα του.

Οι Μηχανικοί Λογισμικού, διακρίνουν το ελάττωμα λογισμικού (software fault) από την αποτυχία λογισμικού (software failure). Στην περίπτωση της αποτυχίας, το λογισμικό δεν ανταποκρίνεται στις απαιτήσεις του χρήστη. Το ελάττωμα είναι ένα προγραμματιστικό λάθος το οποίο μπορεί (λόγω μη προβλέψιμων ή αστάθμητων παραγόντων) να οδηγήσει σε αποτυχία. Λόγοι που μπορεί να οδηγήσουν το ελάττωμα σε αποτυχία είναι όταν το λογισμικό εφαρμοστεί σε μια διαφορετική πλατφόρμα λογισμικού ή ένα διαφορετικό μεταγλωττιστή, ή όταν το λογισμικό απλά επεκταθεί. Ένα ελάττωμα μπορεί επίσης να περιγραφεί ως ένα λάθος σημασιολογικής ορθότητας του προγράμματος.

Ο έλεγχος λογισμικού αποτελεί αναπόσπαστο κομμάτι της διαδικασίας διασφάλισης ποιότητας του (SQA - software quality assurance). Μπορεί όμως και να θεωρηθεί ως μια αυτόνομη εργασία. Στην πράξη πολλές εταιρείες δεν εφαρμόζουν τεχνικές SQA αλλά εντούτοις προβαίνουν σε διεξοδικό έλεγχο των προϊόντων τους.

2.2. Σύντομη ιστορική αναδρομή ελέγχου

Ο ορισμός έλεγχος λογισμικού έκανε την εμφάνιση του το 1979, όταν ο Glenford Myers, μέσω του βιβλίου «The art of software testing» [6] έδωσε τον πρώτο ορισμό για τον έλεγχο:

> *«Έλεγχος είναι η διαδικασία της εκτέλεσης ενός προγράμματος ή συστήματος με σκοπό την ανεύρεση λαθών»*

Ο πιο πάνω ορισμός ήταν ο καλύτερος διαθέσιμός που υπήρχε τον καιρό εκείνο, μέχρι να γραφτεί το βιβλίο του. Έτσι τον καιρό εκείνο ο έλεγχος ξεκινούσε όταν τελείωνε ο κύκλος παραγωγής λογισμικού και ο κυριότερος στόχος τους ήταν η ανεύρεση λαθών.

Το 1988 ο Bill Hetzel [7], στο βιβλίο του έδωσε μια διαφορετική ερμηνεία στον έλεγχο λογισμικού, η οποία είχε προσθέσει και την έννοια της ποιότητας του λογισμικού. Ο ορισμός που έδωσε ήταν ο εξής:

> *«Έλεγχος είναι οποιαδήποτε ενέργεια που στόχο έχει να αξιολογήσει μια παράμετρο του συστήματος ή προγράμματος. Έλεγχος είναι η μέτρηση της ποιότητας του λογισμικού»*

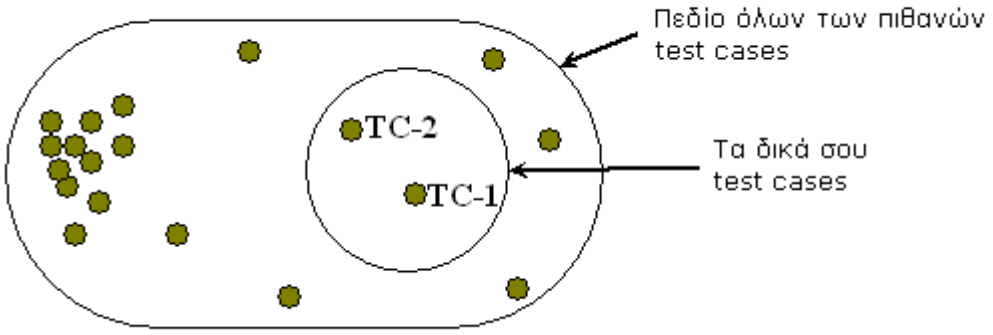
Και οι δύο ορισμοί είναι ορθοί μέχρι σήμερα, αφού ο κάθε ένας βλέπει τον έλεγχο λογισμικού από μια συγκεκριμένη οπτική γωνία. Το πρόβλημα που τους χαρακτηρίζει είναι η εμβέλεια και για τον λόγο αυτό το 2002 είχε διατυπωθεί ένας νέος ορισμός:

> *«Έλεγχος είναι η ταυτόχρονη διαδικασία κύκλου ζωής της επιστήμης, χρήσης και συντήρησης υλικού ελέγχου με σκοπό την μέτρηση και βελτίωση της ποιότητας του λογισμικού που ελέγχεται»*

2.3. Ανάλυση ρίσκου

Είναι αδύνατο να εγγυηθούμε ότι ένα σύστημα λογισμικού θα είναι τέλειο, επειδή αποτυχίες μπορούν να προκύψουν από πολλούς απρόβλεπτους παράγοντες. Ένα λάθος που βρίσκεται κρυμμένο σε ένα λογισμικό το οποίο λειτουργούσε ομαλά για πολλά χρόνια, μπορεί απρόβλεπτα να προκαλέσει δυσλειτουργία του συστήματος. Ένα σύστημα μπορεί να σταματήσει να λειτουργεί όταν οι ατέλειες του παραμένουν κρυμμένες για χρόνια και εμφανίζονται ξαφνικά. Αυτή η απρόβλεπτη εμφάνιση αποτυχίας, μπορεί να είναι το αποτέλεσμα αλλαγών σε πρωτόκολλα, η ασυμβατότητα διασυνδέσεων επικοινωνίας (interfaces) σε κάποιο μέρος ενός συστήματος όταν αυτό χρειαστεί να επικοινωνήσει με ένα «γερασμένο» σύστημα (legacy system), η αύξηση του αριθμού χρηστών που μπορεί να στρεσάρουν το σύστημα, οι αλλαγές που επήλθαν στο μοντέλο εργασίας (business model) μιας επιχείρησης το οποίο μπορεί να οδηγήσει στην χρησιμοποίηση του συστήματος κατά τρόπο ο οποίος δεν είχε χρησιμοποιηθεί μέχρι τότε ή ακόμη λόγω μίας αλλαγής στο λειτουργικό σύστημα [8].

Οι επιστήμονες της Μηχανικής Λογισμικού και οι υπεύθυνοι ελέγχου, έχουν αντιληφθεί ότι είναι αδύνατον να ελεγχθούν τα πάντα ακόμα και στα πιο συνηθισμένα συστήματα. Τα χαρακτηριστικά και οι παράμετροι μιας απλής εφαρμογής έχουν ως επακόλουθο εκατομμύρια συνδυασμών εισόδων, οι οποίες θα έπρεπε να υλοποιηθούν σε δοκιμές ελέγχου (test cases), κάτι που όπως μπορούμε να αντιληφθούμε είναι αδύνατον. Στις περισσότερες περιπτώσεις το **τι** εξετάζεις είναι σημαντικότερο από το **πόσο** εξετάζεις ένα σύστημα.



Σχήμα 2.1: Πεδίο πιθανών δοκιμών ελέγχου σε ένα σύστημα λογισμικού

Σκοπός της ανάλυσης ρίσκου λογισμικού είναι ο προσδιορισμός του τι πρέπει να ελεγχθεί, τις προτεραιότητες και το βάθος του ελέγχου. Σε μεμονωμένες περιπτώσεις περιλαμβάνει και το τι **δεν** πρέπει να ελεγχθεί. Μια ανάλυση ρίσκου βοηθά επίσης τους υπεύθυνους ελέγχου, να αναγνωρίσουν εφαρμογές ψηλού ρίσκου, οι οποίες θα πρέπει να ελεγχθούν εξαντλητικά και πιθανών συστατικών τα οποία είναι πιθανότερο να παρουσιάσουν λάθη σε σχέση με άλλα. Τα αποτελέσματα της ανάλυσης ρίσκου χρησιμοποιούνται κατά τον σχεδιασμό ελέγχου, για προσδιορισμό των προτεραιοτήτων ελέγχου του λογισμικού που θα ελεγχθεί. Το ιδανικότερο, είναι αυτή η ανάλυση, να γίνεται από μία ομάδα εμπείρων που ανήκουν σε διαφορετικές ομάδες μέσα στον οργανισμό. Υποψήφιοι μπορεί να είναι οι τελικοί χρήστες, προγραμματιστές, πελάτες, τα άτομα που συμβάλλουν στην προώθηση του προϊόντος και άλλοι ενδιαφερόμενοι αφού ο κάθε υποψήφιος μπορεί να συμβάλει με τον δικό του τρόπο. Η ανάλυση ρίσκου συνιστάται να εκτελείται όσο πιο γρήγορα γίνεται ακόμη και μόλις μαζευτούν οι πρώτες απαιτήσεις του συστήματος.

2.4. Τεχνικές ελέγχου

Πριν παραδοθούν στον τελικό χρήστη, όλα τα λειτουργικά συστήματα υφίστανται κάποιο έλεγχο αξιοπιστίας για να διαπιστωθεί κατά πόσο ικανοποιούν τις απαιτήσεις και τις προδιαγραφές τους. Ο έλεγχος Μαύρου Κουτιού (black box) και ο έλεγχος Άσπρου Κουτιού (white box) είναι μέθοδοι για έλεγχο ορθότητας μιας εφαρμογής, ανάλογα με

το αν αυτός που διεξάγει τον έλεγχο δεν γνωρίζει ή γνωρίζει αντίστοιχα, εσωτερικές λεπτομέρειες του συστήματος. Στην τεχνική του black box μεταχειριζόμαστε το σύστημα ως ένα «μαύρο κουτί», έτσι δεν χρειάζεται γνώση της εσωτερικής δομής, αλλά εστιάζεται συγκεκριμένα στην χρησιμοποίηση της εσωτερικής γνώσης του λογισμικού για να καθοδηγήσει την επιλογή των test data (δεδομένων ελέγχου). Στην τεχνική του white box, ο ελεγκτής πρέπει να γνωρίζει την εσωτερική δομή του προγράμματος και επιλέγει τις δοκιμές ελέγχου με βάση τον κώδικα της εφαρμογής. Για να ελεγχτεί καλά ένα σύστημα πρέπει να ακολουθηθούν και οι δύο προσεγγίσεις.

2.4.1. Έλεγχος Μαύρου Κουτιού - Black Box Testing

Ο Black-box έλεγχος είναι επίσης γνωστός και ως functional έλεγχος ή concrete box έλεγχος. Είναι μια τεχνική ελέγχου λογισμικού όπου η εσωτερική δομή του προγράμματος που ελέγχεται δεν είναι γνωστή στον ελεγκτή. Για παράδειγμα, στο Black Box Testing ο ελεγκτής γνωρίζει μόνο το input και το επιθυμητό αποτέλεσμα αλλά δεν γνωρίζει πως το πρόγραμμα εξάγει αυτό το αποτέλεσμα. Ο συγκεκριμένος ελεκτής δεν εξετάζει ποτέ προγραμματιστικό κώδικα (programming code) και δεν χρειάζεται περισσότερη γνώση του προγράμματος εκτός από τις προδιαγραφές (specifications). Στην τεχνική αυτή η προσοχή του ελεγκτή εστιάζεται κυρίως στις λειτουργικές απαιτήσεις του προϊόντος (στις προδιαγραφές και τις λειτουργίες του) και όχι στον κώδικα.

Ο όρος black box δείχνει ότι η εσωτερική υλοποίηση του προγράμματος δεν εξετάζεται από τον ελεγκτή, δηλαδή το σύστημα χρησιμοποιείται ως ένα μαύρο κουτί (βλέπε σχήμα 2.2). Για τον λόγο αυτό ο έλεγχος αυτός δεν γίνεται κατ' ανάγκη από τον προγραμματιστή. Στο σύστημα δίνεται μία είσοδος-ερέθισμα και αν το αποτέλεσμα είναι αυτό που αναμένεται τότε η συγκεκριμένη φάση ελέγχου θεωρείται επιτυχημένη.

Τα πλεονεκτήματα και μειονεκτήματα της μεθόδου αυτής ακολουθούν:

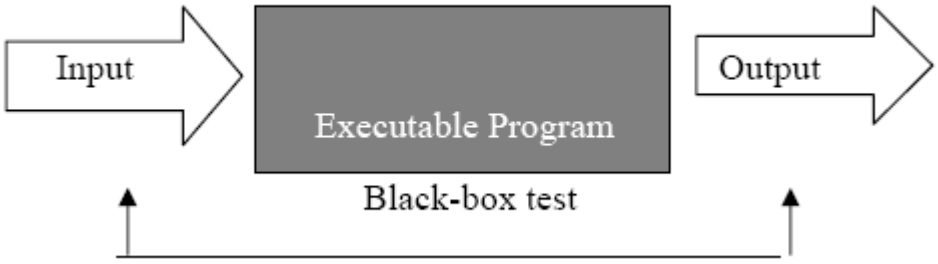
Πλεονεκτήματα:

- Ο ελεγκτής δεν χρειάζεται να έχει οποιαδήποτε γνώση της υλοποίησης, συμπεριλαμβανομένων συγκεκριμένων γλωσσών προγραμματισμού.
- Είναι αποτελεσματικότερος σε μεγάλα κομμάτια κώδικα σε σχέση με το glass box testing.
- Ο προγραμματιστής και ο ελεγκτής είναι ανεξάρτητοι ο ένας απ' τον άλλο.
- Οι έλεγχοι γίνονται από την πλευρά άποψης του χρήστη.
- Βοηθά στο να ανακαλυφθούν οποιεσδήποτε ασάφειες ή ασυνέπειες στις προδιαγραφές.

- Τα test cases μπορούν να σχεδιαστούν μόλις ολοκληρωθούν οι προδιαγραφές.

Μειονεκτήματα:

- Χωρίς σαφείς και συνοπτικές προδιαγραφές, οι περιπτώσεις δοκιμής (test cases) είναι δύσκολο να σχεδιαστούν.
- Μόνο ένας μικρός αριθμός πιθανών inputs μπορεί πραγματικά να ελεγχθεί (το να δοκιμάσουμε κάθε πιθανό input stream θα διαρκούσε σχεδόν για πάντα).
- Μπορεί να υπάρξει περιττή επανάληψη των test inputs, αν ο ελεγκτής δεν ενημερώνεται για τα test cases του προγραμματιστή.
- Δεν μπορεί να κατευθυνθεί προς συγκεκριμένα τμήματα του κώδικα που μπορούν να είναι πολύ σύνθετα και επομένως επιτρέπονται περισσότερα λάθη.
- Μπορεί να αφήσει πολλά μονοπάτια του προγράμματος ανεξέταστα.
- Οι περισσότερες έρευνες που αφορούν τον έλεγχο έχουν κατευθυνθεί προς την τεχνική του glass box testing.



Σχήμα 2.2: Έλεγχος Μαύρου Κουτιού: Λαμβάνει υπόψη μόνο το input και το output του προγράμματος δίχως να λάβει υπόψη το εσωτερικό του μέρος.

Στην συνέχεια αναφέρω μερικά είδη ελέγχου, τα οποία ανήκουν στην κατηγορία του black box testing:

- **Διαίρεση ισοδυναμίας (Equivalence partitioning):** Η τεχνική αυτή σχεδιάζεται για να ελαχιστοποιήσει τον αριθμό των δοκιμών ελέγχου που θα χρειαστούν για τον έλεγχο του συστήματος. Ουσιαστικά με την τεχνική αυτή ομαδοποιούνται διάφορες τιμές εισόδου, οι οποίες παράγουν το ίδιο αποτέλεσμα εξόδου. Οι υπεύθυνοι της τεχνικής αυτής πρέπει να είναι ιδιαίτερα προσεκτικοί, έτσι ώστε να επιλέξουν τα σωστά κομμάτια κατά την διαίρεση, για να είναι ακριβής και αποδοτικός ο έλεγχος και να μην υπάρχουν περιπτώσεις δοκιμών ελέγχου που να μην ελεγχθούν. Η μέθοδος αυτή έχει κάποια μειονεκτήματα όπως το ότι υπάρχει περίπτωση να μην εξεταστούν όλα τα δεδομένα εισόδου και ότι δεν υπάρχουν κάποιες οδηγίες (κατευθυντήριες γραμμές) για την επιλογή των δεδομένων εισόδου.

- **Πίνακες αποφάσεων (Decision tables):** Στην τεχνική αυτή, έχουμε πίνακες που περιέχουν όλες τις πιθανές εισόδους και όλες τις πιθανές εξόδους ενός προγράμματος.
- **Ανάλυση ορίων (Boundary value analysis):** Στην τεχνική αυτή εξετάζονται οι οριακές τιμές των δεδομένων εισόδου. Η ανάλυση ορίων είναι σημαντική επειδή συνήθως είναι τα όρια που προκαλούν τα λάθη στην λειτουργία του συστήματος και επομένως το οδηγούν σε αποτυχία. Στην τεχνική αυτή, οι οριακές τιμές των δεδομένων εισόδου χρησιμοποιούνται για την παραγωγή περιπτώσεων δοκιμών ελέγχου για να εξασφαλιστεί η σωστή λειτουργία του συστήματος. Τα πλεονεκτήματα της μεθόδου είναι ότι είναι πολύ καλή στην εύρεση δοκιμών ελέγχου που μπορούν να οδηγήσουν σε αποτυχία του συστήματος, υπάρχουν σαφείς κατευθυντήριες γραμμές για την επιλογή των δοκιμών ελέγχου και ότι είναι μικρό το σύνολο των δοκιμών ελέγχου που παράγονται. Τα μειονεκτήματα είναι ότι δεν ελέγχονται όλα τα πιθανά δεδομένα ελέγχου και ότι δεν εξετάζονται οι πιθανές εξαρτήσεις μεταξύ των συνδυασμών των δεδομένων εισόδου.
- **Διαγράμματα μετάβασης καταστάσεων (State transition diagrams):** Η τεχνική αυτή χρησιμοποιείται από την αρχή της αντικειμενοστρεφούς μοντελοποίησης (object-oriented modeling). Η βασική ιδέα είναι η κατασκευή μιας μηχανής με συγκεκριμένο αριθμό καταστάσεων. Η μηχανή παίρνει κάποια μηνύματα (events) από τον έξω κόσμο και κάθε ένα από αυτά προκαλεί την μετάβαση (transition) της μηχανής από μια κατάσταση σε άλλη. Το μεγαλύτερο μειονέκτημα αυτής της μεθόδου είναι ότι πρέπει να οριστούν όλες οι πιθανές καταστάσεις του συστήματος.
- **Εξερευνητικός έλεγχος (Exploratory testing):** Η μέθοδος αυτή είναι μια αλληλεπιδραστική διαδικασία όπου η εξερεύνηση του προϊόντος, ο σχεδιασμός ελέγχου και η εκτέλεση ελέγχου γίνονται ταυτόχρονα. Δηλαδή διαφέρει από μια διαδικασία στην οποία οι δοκιμές ελέγχου σχεδιάζονται όλες στην αρχή και μετά εκτελούνται. Κατά μια έννοια ότι αυτός που κάνει τον έλεγχο μαθαίνει το προϊόν στην πορεία του ελέγχου. Δίνεται έμφαση στην δημιουργικότητα και στον αυθορμητισμό του εκλεκτή. Βάση των αποτελεσμάτων κάποιων δοκιμών ελέγχου, ο εκλεκτής του κώδικα ωθείται να ελέγξει σε περισσότερο βάθος την συγκεκριμένη περιοχή. Με τον τρόπο αυτό οι δημιουργικές περιοχές που ελέγχονται επεκτείνονται αμέσως λόγω του ότι η έξοδος ενός ελέγχου επηρεάζει τον σχεδιασμό του επόμενου.
- **Ορθογώνιοι πίνακες (Orthogonal arrays):** Πρόκειται για μαθηματικές οντότητες, οι οποίες έχουν την εξής ιδιότητα: αν επιλέξουμε δύο οποιεσδήποτε

στήλες του πίνακα, τότε όλοι οι συνδυασμοί των αριθμών εμφανίζονται σε αυτές τις στήλες. Χρησιμοποιούνται για να επιλεγούν οι δοκιμές ελέγχου.

- **Επιτόπου έλεγχος (Ad hoc testing):** Ο επιτόπου έλεγχος μπορεί να περιγραφεί ως μια εξερευνητική περίπτωση, στην οποία αναμένεται ότι θα τρέξουμε μόνο μια φορά τον κώδικα για να ανακαλύψουμε ένα σφάλμα. Μια σχετικά απλή τεχνική που αρκετοί το βλέπουν ως χάσιμο χρόνου, ενώ άλλοι έμπειροι ελεγκτές βρίσκουν την τεχνική αυτή μια από τις καλύτερες για να την εύρεση συγκεκριμένων ειδών λαθών.
- **Τυχαίος έλεγχος (Random testing):** Με την τεχνική αυτή τα δεδομένα ελέγχου δημιουργούνται τυχαία, συνήθως με την βοήθεια κάποιου εργαλείου.
- **Ημι-τυχαίος έλεγχος (Semi random testing):** Η τεχνική αυτή χρησιμοποιείται για την μείωση των συνδυασμών εισόδου και ακολούθως εφαρμόζεται η τεχνική random testing.

2.4.2. Έλεγχος Άσπρου Κουτιού – White Box Testing

Ο White - box έλεγχος είναι επίσης γνωστός και ως glass box, structural, clear box and open box testing. Είναι μια τεχνική ελέγχου όπου ακριβής γνώση των εσωτερικών λειτουργιών του κώδικα που εξετάζεται πρέπει να είναι εις γνώση του ελεγκτή για να επιλέξει τα test data. Ο έλεγχος είναι σωστός μόνο αν ο ελεγκτής γνωρίζει τι ακριβώς πρέπει να κάνει το πρόγραμμα. Έτσι ο ελεγκτής μπορεί μετά τον έλεγχο να δει αν τα αποτελέσματα αποκλίνουν από τον προτιθέμενο στόχο.

Ονομάζεται glass box testing γιατί εξετάζεται η εσωτερική υλοποίηση του προγράμματος από τον ελεγκτή, δηλαδή μπορεί να δει καθαρά το εσωτερικό του κουτιού. Το όνομα structural testing, δείχνει ότι οι δοκιμές ελέγχου σχεδιάζονται με βάση την εσωτερική δομή και λογική του προγράμματος. Ο έλεγχος γίνεται με βάση τον κώδικα με αποτέλεσμα να έχουν μεγαλύτερη πιθανότητα εντοπισμού λάθη από την πλευρά του προγραμματιστή.

Παραδείγματα της τεχνικής αυτής είναι οι μέθοδοι: path coverage, statement coverage, branch coverage, condition coverage, edge coverage, κ.λπ. που θα παρουσιαστούν αναλυτικά σε παρακάτω παράγραφο. Οι δοκιμές ελέγχου παράγονται προσεκτικά βάση ενός κριτήριου κάλυψης. Για μια τιμή εισόδου που δίνεται στο σύστημα ελέγχεται όχι μόνο το αν δημιουργείται το σωστό αποτέλεσμα, αλλά και το πώς προκύπτει.

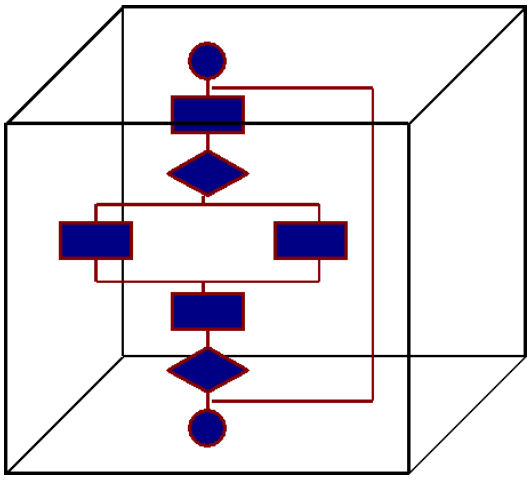
Τα πλεονεκτήματα και μειονεκτήματα της μεθόδου αυτής ακολουθούν:

Πλεονεκτήματα:

- Με την τεχνική, ελέγχεται πως προκύπτει ένα αποτέλεσμα και όχι μόνο αν είναι ορθό. Με αυτό τον τρόπο μπορούμε να βρούμε περιπτώσεις όπου μπορεί να παραχθεί σωστό αποτέλεσμα, το οποίο όμως παράγεται για λάθος λόγους ή λόγο συγκυριών. Το φαινόμενο αυτό ονομάζεται συμπτωματική ακρίβεια (coincidental correctness) και ανακαλύπτεται εύκολα με αυτή την τεχνική.
- 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 του black box testing.
- Αν ένα σύστημα δεν ελεγχθεί βάση των εσωτερικών λειτουργιών του είναι αδύνατο να ελεγχθεί με όλους τους πιθανούς τρόπους που δουλεύει.

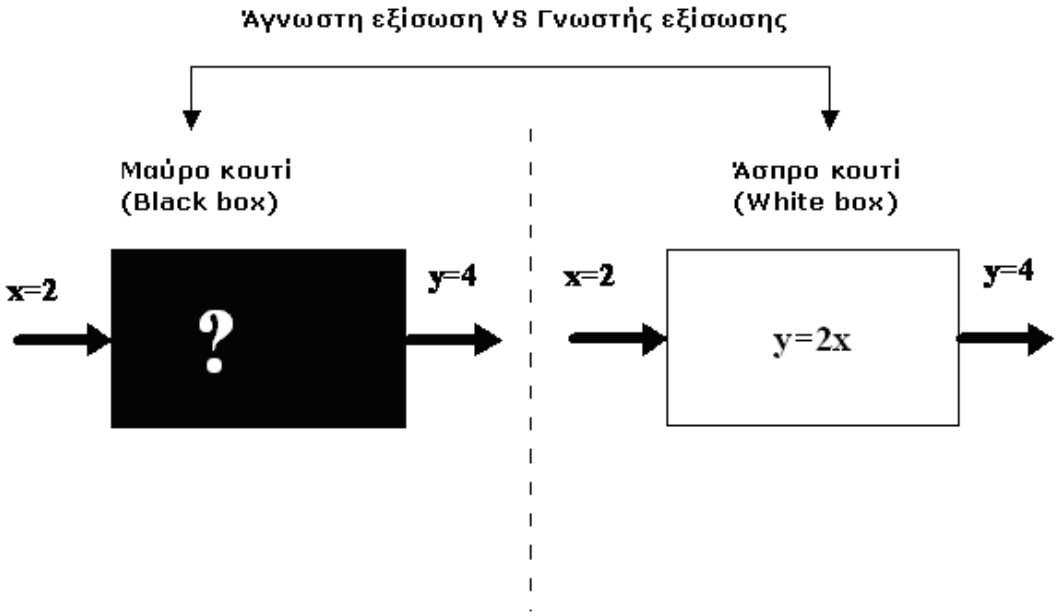
Μειονεκτήματα:

- Είναι σχεδόν αδύνατο σε περιπτώσεις μεγάλων συστημάτων να εξεταστούν όλα τα μονοπάτια του κώδικα έτσι ώστε να ανακαλυφθούν όλα τα λάθη που υπάρχουν. Επιπλέον δεν υπάρχει τρόπος να εντοπισθούν τα μονοπάτια τα οποία έχουν παραληφθεί.
- Με την τεχνική αυτή είναι πολύ δύσκολο να εντοπίσουμε λάθη λογικής.
- Δεδομένου ότι για να γίνει η τεχνική αυτή πρέπει ο ελεγκτής να γνωρίζει τον κώδικα και την εσωτερική δομή του προγράμματος, ένας έμπειρος και εξειδικευμένος ελεγκτής θα έχει μεγαλύτερες οικονομικές απαιτήσεις.



Σχήμα 2.3: Έλεγχος Άσπρου Κουτιού: Λαμβάνει υπόψη την εσωτερική δομή του προγράμματος

Στο επόμενο σχήμα παρουσιάζεται η διαφορά των δυο τεχνικών ελέγχου που αναλύθηκαν πιο πάνω.



Σχήμα 2.4: Έλεγχος Μαύρου κουτιού Vs έλεγχος άσπρου κουτιού

2.4.3. Έλεγχος Γκρίζο κουτί - Gray Box Testing

Το Gray Box Testing είναι μια τεχνική ελέγχου λογισμικού που συνδυάζει την τεχνική του black box testing και white box testing. Το Gray Box Testing δεν είναι black box testing επειδή ο ελεγκτής γνωρίζει λίγες από τις εσωτερικές λειτουργίες του software που εξετάζει. Στο Gray Box Testing ο ελεγκτής εφαρμόζει ένα περιορισμένο αριθμό test cases στις εσωτερικές λειτουργίες του προγράμματος που εξετάζει.

Η τεχνική αυτή έχει ως στόχο της να αξιοποιήσει όσο το δυνατό πιο πολύ τα πλεονεκτήματα του white box και του black box testing έτσι ώστε να πετύχει τη μεγαλύτερη βελτίωση στην ποιότητα του συστήματος. Οι τεχνικές white box και black box testing βελτιώνουν την ποιότητα ενός συστήματος κατά 40%. Μαζί και οι δύο τεχνικές μπορούν να βελτιώσουν την ποιότητα ενός συστήματος κατά 60% [9].

Μερικοί ορισμοί του Gray Box Testing είναι:

*«Οι έλεγχοι εμπλέκουν inputs και outputs, αλλά ο σχεδιασμός ελέγχου είναι συμμορφωμένος από πληροφορίες για τον κώδικα ή την λειτουργία του έτσι ώστε κανονικά θα ήταν εκτός της βλέψης του ελεγκτή.» [**Cem Karner**]*

*«Ο έλεγχος σχεδιάστηκε βασισμένος στην γνώση του αλγορίθμου, των εσωτερικών καταστάσεων, ή άλλων υψηλού επιπέδου περιγραφών της συμπεριφοράς προγράμματος.» [**Doug Hoffman**]*

2.4.4. Πολυπλοκότητα κώδικα - Cyclomatic Complexity

Κυκλωματική πολυπλοκότητα είναι μια μετρική λογισμικού (software metric). Αναπτύχθηκε από τον Thomas McCabe και χρησιμοποιήθηκε για να υπολογίζει την πολυπλοκότητα ενός προγράμματος. Υπολογίζει απευθείας τον αριθμό των γραμμικών ανεξάρτητων μονοπατιών διαμέσου του κώδικα του προγράμματος [10]. Η κυκλωματική πολυπλοκότητα συχνά αναφέρεται και ως πολυπλοκότητα προγράμματος ή ως πολυπλοκότητα του McCabe.

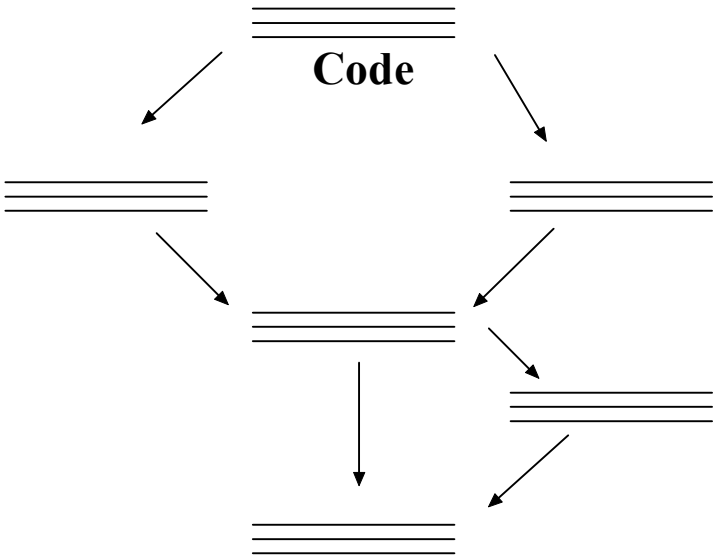
Η κυκλωματική πολυπλοκότητα προήλθε από την μαθηματική θεωρία γράφων. Η πολυπλοκότητα C προσδιορίζεται με την εξής φόρμουλα:

$$C = e - n + 2p$$

όπου: e = ο αριθμός των πλευρών του γράφου ροής
n = ο αριθμός των κόμβων του γράφου και
p = ο αριθμός των ανεξάρτητων διαδικασιών

Η σημασία της μετρικής είναι πολύ σημαντική αφού εξασφαλίζει μέτρηση της πολυπλοκότητας ενός κομματιού κώδικα λογισμικού. Με την μετρική αυτή μπορούμε να προσδιορίσουμε το μέγιστο αριθμό ελέγχων που χρειάζονται για να καλυφθεί το κριτήριο branch coverage.

Παρατηρώντας το πιο κάτω σχήμα βλέπουμε ότι υπάρχουν τέσσερα μονοπάτια τα οποία μπορούμε να ακολουθήσουμε για να φτάσουμε στο τελευταίο κομμάτι κώδικα. Άρα για να καλύψουμε το κριτήριο Path coverage θα πρέπει να δημιουργήσουμε τουλάχιστον τέσσερα test cases. Αν εφαρμόσουμε τον προτεινόμενο τύπο τότε θα έχουμε C = 7 – 6 + 2(1) = 3. Ο τύπος μας δίνει την πολυπλοκότητα του πιο κάτω κώδικα αλλά πιο σημαντικό είναι ότι η πολυπλοκότητα αυτή είναι ο απαιτούμενος αριθμός test cases για να καλύψουμε το branch coverage κριτήριο.



Σχήμα 2.5: Παράδειγμα κώδικα – Χρήση Μετρικής McCabe

Η βαθμολογία που δίνει η πιο κάτω μετρική για μία ρουτίνα είναι ο αριθμός των σημείων αποφάσεων +1. Σημεία αποφάσεων θεωρούνται τα εξής: If, while, repeat, for, and, or και κάθε κόμβος (branch) του case statement.

Ανάλογα με τον αριθμό που υπολογίζουμε για την κυκλωματική πολυπλοκότητα μπορούμε να δούμε ποια είναι η ανάλυση ρίσκου του προγράμματος ανάλογα μέσα σε ποιο διάστημα threshold βρισκόμαστε (βλέπε πιο κάτω πίνακα).

Πίνακας : Πολυπλοκότητα κώδικα	
Πολυπλοκότητα κώδικα	**Αποτίμηση ρίσκου**
1-10	Απλό πρόγραμμα, χωρίς πολύ ρίσκο
11-20	Πιο περίπλοκο, μέτριο ρίσκο
21-50	Περίπλοκο, μεγάλου ρίσκου πρόγραμμα
μεγαλύτερο από 50	Πολύ υψηλού ρίσκου πρόγραμμα

Πίνακας 2.1: Πολυπλοκότητα κώδικα

Για παράδειγμα το πιο πάνω πρόγραμμα που αναλύσαμε πέφτει στην πρώτη κατηγορία και άρα είναι απλό πρόγραμμα, χωρίς πολύ ρίσκο.

2.4.5. Στατικός έλεγχος – Statistic Testing

Ο στατικός έλεγχος είναι ένα είδος ελέγχου λογισμικού όπου το λογισμικό δεν εκτελείται εκείνη την στιγμή. Αυτό έρχεται σε αντίθεση με τον δυναμικό έλεγχο. Είναι γενικά μη λεπτομερής έλεγχος, αλλά ελέγχει κυρίως τη λογική του κώδικα ή του αλγορίθμου. Θεωρείται κυρίως συντακτικός έλεγχος (syntax checking) του

κώδικα ή χειρονακτικό διάβασμα (manually reading) του κώδικα για να βρεθούν λάθη. Αυτός ο τύπος ελέγχου μπορεί να χρησιμοποιηθεί από τον προγραμματιστή [11].

Οι στατικοί έλεγχοι ενός συστήματος είναι δηλαδή ανεξάρτητοι του χρόνου και δεν χρειάζονται να εκτελεστεί το πρόγραμμα για να το ελέγξουν. Παραδείγματα τέτοιων ελέγχων αποτελούν οι έλεγχοι ροής δεδομένων (flow control) και οι συντακτικοί έλεγχοι (syntax checking).

2.4.6. Δυναμικός Έλεγχος - Dynamic Testing

Ο δυναμικός έλεγχος είναι ένας όρος που χρησιμοποιείται στην τεχνολογία λογισμικού για να περιγράψει τον έλεγχο μιας δυναμικής συμπεριφοράς κώδικα. Γι' αυτό και η δυναμική ανάλυση αναφέρεται στην εξέταση της φυσικής απόκρισης του συστήματος για μεταβλητές που δεν είναι σταθερές και αλλάζουν με τον χρόνο. Στον δυναμικό έλεγχο το λογισμικό πρέπει να γίνει compile και run σε αντίθεση με το static testing. Μερικές δυναμικές μεθοδολογίες ελέγχου περιλαμβάνουν το unit testing, integration testing, system testing και acceptance testing (περιγράφονται πιο κάτω).

2.5. Επίπεδα και φάσεις του ελέγχου λογισμικού

2.5.1. Έλεγχος μονάδων – Unit Testing

Το unit testing είναι μια διαδικασία που χρησιμοποιείται να επιβεβαιώσει ότι ανεξάρτητα units δουλεύουν σωστά. Το unit είναι η μικρότερη μονάδα ελέγχου σε μια εφαρμογή. Με άλλα λόγια ένα unit είναι ένα συστατικό (component) το οποίο δεν μπορεί να διαιρεθεί σε αλλά συστατικά.

Οι μηχανικοί λογισμικού γράφουν τα test cases (βάση του white box testing) και εξετάζουν κατά πόσο το unit λειτουργεί σωστά (με βάση τον κώδικα). Ο έλεγχος unit είναι σημαντικός ώστε να βεβαιωθούμε ότι ο κώδικας είναι ορθός (δεν έχει λάθη), πριν τον ενώσουμε με τα υπόλοιπα συστατικά του πλήρους προγράμματος. Σε περίπτωση που το unit ενωθεί με τον υπόλοιπο κώδικα, ενώ υπάρχει κάποιο λάθος (failure), θα είναι πολύ πιο δύσκολο να βρεθεί. Περίπου το 65% από όλα τα bugs που μπορούν να εντοπιστούν, εντοπίζονται με το unit testing [12].

Ο στόχος του unit testing είναι να απομονώσει κάθε κομμάτι κάποιου προγράμματος και να δείξει ότι τα ανεξάρτητα κομμάτια του είναι σωστά. Ένα unit test παρέχει αυστηρό συμβόλαιο ότι το κομμάτι κώδικα που ελέγχεται είναι ορθό. Έτσι σαν αποτέλεσμα έχουμε πολλά πλεονεκτήματα.

2.5.2. Έλεγχος ολοκλήρωσης – Integration testing

Το integration testing είναι μια λογική προέκταση του unit testing. Στην πιο απλή του μορφή είναι όταν δύο units που έχουν ήδη ελεγχθεί, συνδυάζονται σε ένα component και το interface μεταξύ τους εξετάζεται. Το component αναφέρεται σε ένα ολοκληρωμένο άθροισμα περισσότερων από ένα units. Σε ένα ρεαλιστικό σενάριο, πολλά units συνδυάζονται σε components τα οποία στη συνέχεια αθροίζονται σε ακόμα μεγαλύτερα κομμάτια προγράμματος. Η ιδέα εδώ είναι να συνδυάζονται και να ελέγχονται συνδυασμοί κομματιών και τελικά να επεκτείνεται η διαδικασία και στο τέλος να εξετάζονται τα modules με τα modules άλλων groups. Στο τέλος όλα τα modules κάνουν μια διαδικασία και ελέγχονται μαζί.

Τα test cases δημιουργούνται εξετάζοντας τα interfaces των διαφόρων units της εφαρμογής. Τα test cases αυτά μπορεί να δημιουργηθούν με την τεχνική του black box testing όταν ο ελεγκτής διαπιστώσει ότι ένα test case απαιτεί την αλληλεπίδραση πολλών μονάδων. Αλλιώς, δημιουργούνται τα test cases βάση της τεχνικής του white box testing, τα οποία μπορεί να βρει ο ελεγκτής εξετάζοντας τα interfaces.

2.5.3 Λειτουργικός Έλεγχος ή Έλεγχος Συστήματος – Functional/System Testing

Ο Functional ή System έλεγχος ενός λογισμικού γίνεται σε ένα τελειωμένο, ολοκληρωμένο σύστημα για να αποτιμηθεί η συμμόρφωση του συστήματος με τις καθορισμένες απαιτήσεις. Ο έλεγχος του συστήματος (system testing), περιλαμβάνει την εγκατάσταση του συστήματος σε διάφορα περιβάλλοντα, για να βεβαιωθούμε ότι το πρόγραμμα δουλεύει στα διάφορα περιβάλλοντα του χρήστη, με διάφορα versions και τύπους λειτουργικών συστημάτων ή/και εφαρμογές.

Αυτό το είδος ελέγχου χρησιμοποιεί την τεχνική του black box testing, αφού οι ελεγκτές εξετάζουν τον σχεδιασμό σε ψηλότερα επίπεδα και τις απαιτήσεις του χρήστη για να σχεδιάσουν τα test cases. Ο λειτουργικός έλεγχος και ο έλεγχος του συστήματος είναι καλύτερα να γίνονται από κάποιον που έχει μια ανεξάρτητη άποψη για το σύστημα (για παράδειγμα ο προγραμματιστής πρέπει να αποκλείεται).

- **Στρεσάρισμα συστήματος (stress testing):** Με τον έλεγχο αυτό, αποτιμάται το σύστημα ή ένα component μέχρι ή και πάνω από τα όρια που καθορίζονται στις προδιαγραφές και στις απαιτήσεις του. Για παράδειγμα αν μια ομάδα αναπτύσσει ένα προϊόν λογισμικού, το οποίο να τρέχει ταμειακές μηχανές, μπορεί να εκφραστεί ότι ο server θα μπορεί να χειρίζεται μέχρι και 30 ταμειακές μηχανές που θα αναζητούν τιμές ταυτόχρονα. Κατά το stress testing μπορεί να δημιουργήσει ένα δωμάτιο με 30 πραγματικές ταμειακές, οι οποίες τρέχουν αυτόματα συναλλαγές επαναληπτικά για κάποιο αριθμό ωρών.

Υπάρχει επίσης η πιθανότητα να υπάρχουν περισσότερες από 30 ταμειακές μηχανές στο δωμάτιο για να δούμε αν το σύστημα μπορεί να ξεπεράσει τις καθορισμένες προδιαγραφές του.

- **Έλεγχος απόδοσης (performance testing):** Με τον έλεγχο αυτό, αποτιμάται η συμμόρφωση του συστήματος ή ενός component με συγκεκριμένες απαιτήσεις απόδοσης. Για το ίδιο παράδειγμα που δόθηκε στον stress testing, μια απαίτηση απόδοσης μπορεί να είναι ότι πρέπει να βρίσκουμε την τιμή που ψάχνουμε σε λιγότερο από 1 δευτερόλεπτο. Με τον έλεγχο απόδοσης (performance testing), ελέγχουμε κατά πόσο το σύστημα μπορεί να βρει τιμές σε λιγότερο από 1 δευτερόλεπτο (ακόμα κι αν τρέχουν 30 ή περισσότερες ταμειακές μηχανές ταυτόχρονα).
- **Έλεγχος ευχρηστίας (usability testing):** Με τον έλεγχο αυτό, αποτιμάται ο βαθμός στον οποίο ένας χρήστης μπορεί να μάθει να χειρίζεται, να ετοιμάζει τα δεδομένα εισόδου για το σύστημα ή για ένα component και να ερμηνεύει τα δεδομένα εξόδου του συστήματος ή του component. Οι stress και performance testing μπορούν να γίνουν, και συνήθως γίνονται αυτόματα, ενώ ο usability testing γίνεται με αλληλεπίδραση του χρήστη με τον υπολογιστή, όπου και παρατηρείται η ευχρηστία του συστήματος.

2.5.4. Έλεγχος αποδοχής – Acceptance Testing

Όταν τελειώσουν οι έλεγχοι από τους δημιουργούς του λογισμικού, το προϊόν παραδίδεται στον πελάτη, και ο πελάτης τρέχει κάποιους black box ελέγχους αποδοχής, βασισμένους στις προσδοκίες τις λειτουργικότητάς του προϊόντος. Ο έλεγχος αποδοχής (acceptance testing), καθορίζει κατά πόσον το σύστημα ικανοποιεί ή όχι τα κριτήρια αποδοχής (κριτήρια που το σύστημα πρέπει να ικανοποιεί για να γίνει αποδεκτό από τον πελάτη) και κατά πόσο ο πελάτης θα αποδεχτεί ή όχι το σύστημα. Τα test cases αυτά, συνήθως καθορίζονται από πριν από τον πελάτη και δίνονται στην ομάδα ελέγχου για να τα τρέξει πριν επιχειρήσει να παραδώσει το προϊόν. Ο πελάτης έχει το δικαίωμα να αρνηθεί την παράδοση του συστήματος αν τα acceptance test cases δεν περάσουν (δηλαδή τα αποτελέσματα δεν είναι σωστά).

2.5.5. Επανέλεγχος – Regression Testing

Στον regression testing γίνεται ένας επιλεκτικός επανέλεγχος του συστήματος ή συγκεκριμένων components του συστήματος για να επιβεβαιωθεί ότι οι τροποποιήσεις που έγιναν στο σύστημα δεν έφεραν κάποιες ανεπιθύμητες αλλαγές, με αποτέλεσμα να μην τηρούνται οι αρχικές απαιτήσεις. Ουσιαστικά αυτό το είδος ελέγχου, εκτελείται κατά

τη φάση της συντήρησης του λογισμικού: μετά από αλλαγές, διορθώσεις ή ενημερώσεις κομματιών του συστήματος. Όπως και στο integration testing, έτσι και στο regression testing, τα test cases μπορούν να δημιουργηθούν μέσω της τεχνικής του black box testing, ή μέσω της τεχνικής του white box testing ή με ένα συνδυασμό των δύο τεχνικών.

2.5.6. Beta Testing

Όταν ένα κομμάτι του συστήματος ή ολόκληρη η έκδοση του συστήματος είναι διαθέσιμη, τότε ο οργανισμός που αναπτύσσει το προϊόν, μπορεί να το προσφέρει δωρεάν σε ένα ή περισσότερους χρήστες τους λεγόμενους beta ελεγκτές. Οι χρήστες αυτοί εγκαθιστούν το λογισμικό και το χρησιμοποιούν όπως θέλουν, με την δέσμευση όμως ότι θα αναφέρουν στον οργανισμό που αναπτύσσει το προϊόν, όσα λάθη ανακαλύπτουν κατά την χρήση του. Οι χρήστες αυτοί επιλέγονται συνήθως λόγω του ότι είναι έμπειροι σε προηγούμενες εκδόσεις (versions) του προϊόντος, ή είναι έμπειροι σε ανταγωνιστικά προϊόντα. Η συγκεκριμένη μέθοδος παρουσιάζει συγκεκριμένα πλεονεκτήματα και μειονεκτήματα.

Πλεονεκτήματα:

- Χαμηλό κόστος αφού οι beta testers γενικά παίρνουν δωρεάν το λογισμικό αλλά δεν πληρώνονται για τον έλεγχο.
- Μπορούν να βρεθούν μη αναμενόμενα λάθη γιατί οι beta testers χρησιμοποιούν το λογισμικό με απρόβλεπτο τρόπο.
- Ένας ευρύς πληθυσμός αναζητά λάθη σε διάφορα περιβάλλοντα (διαφορετικά λειτουργικά συστήματα με διάφορες υπηρεσίες και με μεγάλου πλήθους εφαρμογές να τρέχουν).

Μειονεκτήματα:

- Χαμηλή ποιότητα αναφοράς λαθών λόγω του ότι οι χρήστες μπορεί στην πραγματικότητα να μην αναφέρουν τα λάθη ή μπορεί τα λάθη να αναφέρονται χωρίς αρκετές λεπτομέρειες.
- Υπάρχει έλλειψη συστηματικού ελέγχου επειδή οι χρήστες χρησιμοποιούν το προϊόν με όποιο τρόπο θέλουν.
- Είναι απαραίτητη μεγάλη προσπάθεια για να εξεταστεί μια αναφορά λάθους, ιδιαίτερα όταν υπάρχουν πολλοί beta testers.

Τύπος Ελέγχου	Που Βασίζεται ο Έλεγχος	Τρόπος Ελέγχου	Ποιος κάνει τον έλεγχο
Μονάδων (Unit)	Στο Σχεδιασμό Χαμηλού Επιπέδου, Στην Δομή Κώδικα	Άσπρο Κουτί	Προγραμματιστής
Ολοκλήρωσης (integration)	Στο Σχεδιασμό Χαμηλού Επιπέδου, Στο Σχεδιασμό Υψηλού Επιπέδου	Άσπρο Κουτί / Μαύρο Κουτί	Προγραμματιστής
Λειτουργικός / Συστήματος (Functional/System)	Στο Σχεδιασμό Υψηλού Επιπέδου, Στην Ανάλυση Προδιαγραφών	Μαύρο Κουτί	Ανεξάρτητος ελεγκτής
Αποδοχής (acceptance)	Στην Ανάλυση Προδιαγραφών	Μαύρο Κουτί	Πελάτης
Beta	Ad Hoc	Μαύρο Κουτί	Πελάτης
Επανέλεγχος (Regression)	Στις αλλαγές των Προδιαγραφών, Στο Σχεδιασμό Υψηλού Επιπέδου	Άσπρο Κουτί / Μαύρο Κουτί	Προγραμματιστής ή ανεξάρτητος ελεγκτής

Πίνακας 2.2: Σύνοψη των 6 επιπέδων ελέγχου λογισμικού

2.6. Υλοποίηση ελέγχου

Η υλοποίηση ελέγχου είναι η διαδικασία κατά την οποία δημιουργούνται ή ανακαλύπτονται τα δεδομένων ελέγχου, αποφασίζονται οι μέθοδοι ελέγχου, προετοιμάζεται το περιβάλλον ελέγχου και επιλέγονται ή υλοποιούνται τα εργαλεία που θα χρησιμοποιηθούν για να διευκολύνουν την διαδικασία αυτή. Κατά την διαδικασία αυτή οι υπεύθυνοι ελέγχου έρχονται αντιμέτωποι με τα εξής ερωτήματα:

- Τι εγκατάσταση θα χρειαστεί για το περιβάλλον ελέγχου.
- Πως θα αποκτηθούν τα δεδομένα ελέγχου.
- Ποιες διαδικασίες ελέγχου θα αυτοματοποιηθούν.
- Ποια εργαλεία ελέγχου θα χρησιμοποιηθούν.

- Πως θα επαληθευτούν τα σύνολα ελέγχων.

2.7. Πώς ξέρουμε ότι τέλειωσε ο έλεγχος

Το ερώτημα «πως ξέρουμε ότι έχει τελειώσει ο έλεγχος» είναι πολύ συνηθισμένο. Θα θέλαμε να σκεφτόμαστε ότι αυτό ξεκαθαρίζεται στην έξοδο κριτηρίου για το κάθε επίπεδο (unit, integration, system , acceptance testing). Για παράδειγμα η έξοδος κριτηρίου για αποδοχή ελέγχου (acceptance testing) είναι συνήθως το σημάδι που αναζητούμε και το οποίο δηλώνει ότι ο έλεγχος έχει τελειώσει και το προϊόν είναι έτοιμο για εγκατάσταση και χρησιμοποίηση από τον πελάτη.

Ο Boris Beizer συνόψισε για το πότε πρέπει να σταματάει ο έλεγχος στα εξής:

> *«Δεν υπάρχει ένα απλό, ορθό και λογικό κριτήριο για τερματισμό του ελέγχου. Επιπρόσθετα δεδομένου κάποιων εφαρμόσιμων κριτηρίων, το πόσο σημαντικό είναι το καθένα εξαρτάται πολύ από το προϊόν, το περιβάλλον, τις συνήθειες και τις παραμέτρους ρίσκου.»*

Η συχνότητα ανακάλυψης λαθών χρησιμοποιείται από πολλούς οργανισμούς ως ένα σημαντικό κριτήριο μέτρησης για να τους βοηθήσει στην πρόβλεψη πότε να σταματήσουν το έλεγχο. Πιο συγκεκριμένα όταν η συχνότητα αυτή πέσει κάτω από ένα επίπεδο τότε θεωρείται ότι το προϊόν είναι έτοιμο για να παραδοθεί. Αν και είναι ένα καλό κριτήριο, θα πρέπει να έχουμε υπόψη ότι πιθανόν άλλοι παράγοντες να μπορούν να οδηγήσουν στην πτώση της συχνότητας ανακάλυψης λαθών, όπως η λιγότερη προσπάθεια από την ομάδα ελέγχου ή, η μη δημιουργία νέων test cases, κτλ. Για αυτό συνιστάται να χρησιμοποιείται περισσότερο του ενός κριτηρίου για το τερματισμό του ελέγχου. Τέλος θα πρέπει να αναφέρουμε ότι ενώ όσο ο χρόνος περνάει η συχνότητα ανακάλυψης λαθών μειώνεται αλλά το κόστος ελέγχου αυξάνεται, αφού αυξάνεται και η προσπάθεια ανεύρεσης λαθών.

Μία εμπειρική τεχνική για τερματισμό του ελέγχου είναι με τον υπολογισμό των λαθών που πιθανόν να έχουν μείνει στο πρόγραμμα χρησιμοποιώντας σύγκριση με άλλα παρόμοια συστήματα. Αυτό απαιτεί συλλογή πολλών δεδομένων από το παρελθόν και επίσης τεχνικές κανονικοποίησης για να λαμβάνονται υπόψη οι διαφορές της εμβέλειας, πολυπλοκότητας και ποιότητας του κώδικα.

Η αβεβαιότητα για τον τερματισμό του ελέγχου συνοψίζεται σε μια φράση του Edsger Dijkstra:

> *«Ο έλεγχος ενός προγράμματος μπορεί να χρησιμοποιηθεί για να δείξει την παρουσία λαθών αλλά δεν μπορεί να χρησιμοποιηθεί για να δείξει την απουσία τους!»*
> *("Program testing can be used to show the presence of bugs, but never to show their absence!")*

2.8. Μέτρηση αποτελεσματικότητας ελέγχου

Μια από τις μεγαλύτερες προκλήσεις που αντιμετωπίζουν οι υπεύθυνοι ελέγχου είναι τι μετρικές πρέπει να χρησιμοποιηθούν και να υλοποιηθούν για την μέτρηση της αποδοτικότητας του ελέγχου. Ο τρόπος για μέτρηση της αποτελεσματικότητας του ελέγχου είναι ένα πολύ δύσκολο και σοβαρό σημείο του ελέγχου. Όλες οι μετρικές για αποδοτικότητα του ελέγχου έχουν τα μειονεκτήματα τους.

Γενικά στην βιομηχανία αλλά και στον ακαδημαϊκό τομέα υπάρχουν τρεις μεγάλες κατηγορίες που έχουν δοκιμαστεί για να μετρήσουν την αποδοτικότητα του ελέγχου και είναι οι εξής:

2.8.1. Ικανοποίηση πελάτη

Οι περισσότερες επιχειρήσεις χρησιμοποιούν αυτή τη μέθοδο μέτρησης αφού ο πελάτης είναι ίσως και ο κυριότερος παράγοντας. Έτσι ερευνάται κατά πόσο ο πελάτης είναι ικανοποιημένος από το λογισμικό (ή αλλιώς από το προϊόν) που έχει παραλάβει. Δύο γνωστές μέθοδοι ικανοποίησης πελάτη είναι:

- **Αξιολόγηση μέσω ερωτηματολογίων**: Με αυτή τη μέθοδο ερχόμαστε πιο κοντά στον πελάτη για να μάθουμε πόσο ικανοποιούνται οι ανάγκες του από το λογισμικό που του παραδώσαμε. Παρόλο που είναι μια αντικειμενική μέθοδος, πολύ δύσκολα επιφέρει αποτελεσματικότητα, διότι είναι δύσκολο για κάποιο να ξέρει τι ακριβώς πρέπει να ρωτήσει και πώς να το ρωτήσει για να πάρει σωστά αποτελέσματα.
- **Help desk calls**: Στην μέθοδο αυτή μετρούνται τα τηλεφωνήματα στο help desk για την μέτρηση της ικανοποίησης του πελάτη. Η μέθοδος αυτή αντιμετωπίζει τα ίδια προβλήματα με την προηγούμενη μέθοδο, δηλαδή δεν ξεχωρίζει την ποιότητα του λογισμικού από την αποδοτικότητα της προσπάθειας ελέγχου.

2.8.2. Ατέλειες που βρέθηκαν στον έλεγχο

Η μέτρηση του αριθμού των λαθών που βρέθηκαν τόσο από την διαδικασία ελέγχου, όσο και από τους πελάτες, είναι ένας άλλος παράγοντας μέτρησης της αποδοτικότητας του ελέγχου. Ένα πρόβλημα της μεθόδου αυτής είναι ότι όλα τα λάθη δεν έχουν τον ίδιο βαθμό κρισιμότητας. Είναι σημαντικό τα λάθη να κρίνονται και / ή να χωρίζονται σε κατηγορίες επίδρασης. Ακόμη ένα πρόβλημα αυτής της μεθόδου, είναι ότι ο αριθμός λαθών που υπήρχαν αρχικά, επιδρά σημαντικά στον αριθμό των λαθών που ανακαλύπτονται. Έτσι μετρώντας τον αριθμό των λαθών, η μέθοδος αυτή δεν εστιάζεται

στον έλεγχο και μόνο, αλλά επηρεάζει και την αρχική ποιότητα του λογισμικού που ελέγχεται. Οι τρόποι μέτρησης λαθών είναι οι εξής:

- **Λάθη που βρέθηκαν από πελάτες**

Η μέθοδος αυτή είναι σημαντική αφού τα λάθη αυτά βρέθηκαν από τον πελάτη και προφανώς δεν βρέθηκαν από τον προγραμματιστή. Η μέθοδος αυτή αντιμετωπίζει κάποια προβλήματα όπως το ότι κάποια λάθη είναι πολύ καλά κρυμμένα και είναι δύσκολο να ανακαλυφτούν.

- **Λάθη που βρέθηκαν κατά την διαδικασία ελέγχου και παραγωγής**

Η Defect Removal Efficiency (DRE) είναι μια δυνατή μετρική για την αποδοτικότητα του ελέγχου. Είναι μια υβριδική μέθοδος όπου λαμβάνονται υπόψη τα λάθη που βρέθηκαν κατά την διαδικασία ελέγχου και τα λάθη που βρέθηκαν κατά διάρκεια της παραγωγής του λογισμικού. Η μέθοδος αυτή μετρά πόσα λάθη έχουν βρεθεί από αυτά που θα μπορούσαν να ανακαλυφθούν. Η φόρμουλα υπολογισμού της τεχνικής αυτής είναι η εξής:

$$DRE = \frac{NumberofBugsFoundInTesting}{NumberOfBugsFoundInTesting\ \&\ NumberNotFound}$$

Η μέθοδος αυτή είναι αρκετά καλή αλλά κάποιος που την χρησιμοποιεί πρέπει να έχει υπόψη του κάποια θέματα όπως π.χ. πρέπει να λάβει υπόψη την σοβαρότητα και η κατανομή των λαθών, δεν μπορεί να γνωρίζει αν ο πελάτης του βρήκε όλα τα λάθη, πότε αρχίζει η μέτρηση λαθών και από τι αποτελείται η διαδικασία ανακάλυψης λάθους, και ότι μερικά λάθη δεν μπορούν να βρεθούν.

2.8.3. Κάλυψη λογισμικού

Από τις σημαντικότερες μεθόδους για την μέτρηση της αποδοτικότητας του ελέγχου, θεωρούνται οι μετρήσεις κάλυψης που είναι ανεξάρτητες από την ποιότητα του λογισμικού. Οι ψηλές επιπέδου μετρήσεις κάλυψης π.χ. κάλυψη απαιτήσεων μπορούν να γίνουν μόλις καθοριστούν τα test cases, δηλαδή πριν ακόμη γραφτεί ο κώδικας. Η κάλυψη μπορεί να χρησιμοποιηθεί για την μέτρηση της πληρότητας της ομάδας test cases των ελέγχων τα οποία χρησιμοποιούνται.

- Κώδικα: Πολλοί έμπειροι ελεγκτές πιστεύουν ότι το σημαντικότερο πράγμα για μία ομάδα ελέγχου, είναι η μέτρηση της κάλυψης του κώδικα. Πολλά εργαλεία έχουν ανακαλυφθεί για να βοηθηθεί η συγκεκριμένη μέθοδος. Αυτά τα εργαλεία μετρούν την κάλυψη, δηλαδή πια statement, branch, path, έχουν διαπεραστεί

κατά την εκτέλεση μέσω των test cases που χρησιμοποιήσαμε, από την ποιότητα του λογισμικού.

- Σχεδιασμού: Πρέπει να ελεγχτεί ότι ο σχεδιασμός έχει καλυφτεί πλήρως.
- Απαιτήσεων: Ακόμα ένας σημαντικός παράγοντας μέτρησης της αποδοτικότητας του ελέγχου, είναι η εξέταση του κατά πόσο έχουν υλοποιηθεί όλες οι απαιτήσεις του πελάτη.

2.9. Κριτήρια Επάρκειας Ελέγχου (Test adequacy criteria)

Μια βασική παραδοχή για τον έλεγχο λογισμικού είναι «Δεν υπάρχει λογισμικό χωρίς σφάλματα!!!». Άρα, πετυχημένος έλεγχος είναι ο έλεγχος που εντοπίζει έστω και ένα σφάλμα. Αν δεν εντοπιστούν δεν σημαίνει ότι δεν υπάρχουν αλλά ο έλεγχος πιθανόν να μην ήταν σωστός ή επαρκής. Πως ξέρουμε όμως πότε ο έλεγχος είναι επαρκής; Ένα σημαντικό ερώτημα του ελέγχου λογισμικού είναι πόσο καλά ένα σύνολο τιμών εξετάζει ένα κομμάτι κώδικα. Συνήθως, ο στόχος είναι να ανακαλυφθούν όσο περισσότερα λάθη με την χρήση των κατάλληλων ομάδων τιμών εισόδου με αποτέλεσμα μία ομάδα τιμών εισόδου να θεωρείται καλύτερη από μία άλλη, εάν η πρώτη βρίσκει περισσότερα λάθη σε ένα κομμάτι κώδικα από την δεύτερη. Δυστυχώς, είναι πολύ δύσκολο να προβλεφθεί πόσα λάθη θα ανακαλυφθούν από μία ομάδα test cases και αυτό οφείλεται στη μεγάλη ποικιλία λαθών και στο ότι η σημασία (νόημα) ενός λάθους είναι καθορισμένη με ασάφεια. Έτσι, είναι σημαντικό να καθορίσουμε κριτήρια επάρκειας ελέγχου (test adequacy criteria) ώστε να μπορούμε να αποφασίσουμε κατά πόσο ένα πρόγραμμα έχει ελεγχθεί αρκετά.

Έτσι με βάση το κριτήριο που επιλέχθηκε, πρέπει να δημιουργηθεί το σύνολο τιμών εισόδου για τον κώδικα που πρέπει να ελεγχθεί για πιθανά λάθη. Επειδή η διαδικασία είναι πολύ δύσκολή και χρονοβόρα για να γίνει με το χέρι, οδήγησε στην αυτόματη δημιουργία των ομάδων test cases με την βοήθεια διαφόρων αλγορίθμων που ανακαλύφθηκαν και εξετάζουν την δομή ενός προγράμματος ώστε να δημιουργήσουν επαρκές σύνολα τιμών ελέγχου αυτόματα. Αναφέρονται ενδεικτικά δύο μέθοδοι:

- Τυχαία παραγωγή δεδομένων (Random Test Data Generation)

Σύμφωνα με αυτή την τεχνική, οι τιμές εισόδου για το υπό εξέταση κομμάτι κώδικα παράγονται τυχαία μέχρι να βρεθεί η κατάλληλη τιμή εισόδου. Το πρόβλημα με αυτή την τεχνική είναι ότι για πολύπλοκα προγράμματα ή/και με πολύπλοκα κριτήρια επάρκειας, όπου ένα σύνολο τιμών εισόδου είναι πολύ μικρό για να ικανοποιήσει συγκεκριμένα κριτήρια και το οποίο ανήκει σε ένα μεγάλο πεδίο τιμών, θα έχει πολύ μικρή πιθανότητα να επιλεχθεί.

- Συμβολική παραγωγή δεδομένων (Symbolic Test Data Generation)

Πολλοί μέθοδοι παραγωγής δεδομένων για έλεγχο, χρησιμοποιούν συμβολικές εκτελέσεις για να βρουν τιμές εισόδων και να ικανοποιήσουν μία απαίτηση ελέγχου. Η συμβολική εκτέλεση ενός προγράμματος αποτελείται από ανάθεση συμβολικών τιμών σε μεταβλητές με σκοπό να βρεθεί ένας αφηρημένος μαθηματικός χαρακτηρισμός για το τι ακριβώς κάνει ένα πρόγραμμα. Έτσι το πρόβλημα δημιουργίας δεδομένων ελέγχου απλοποιείται σε ένα πρόβλημα επίλυσης αλγεβρικών εκφράσεων. Ένα από τα προβλήματα που προέκυψαν με την μέθοδο αυτή ήταν όταν σε ένα πρόγραμμα υπήρχαν απροσδιόριστες επαναλήψεις.

2.9.1. Κριτήρια ελέγχου

Ένα από τα βασικά θέματα στο έλεγχο λογισμικού είναι η αντικειμενική εκτίμηση της ποιότητας ελέγχου. Ένα συνηθισμένο ερώτημα που υπήρχε από τα μέσα της δεκαετίας του ‘70 είναι «Τι είναι κριτήριο ελέγχου το οποίο να προσδιορίζει επαρκή συνθήκη ελέγχου;» [20]. Από το 1970 μέχρι σήμερα προτάθηκαν και διερευνήθηκαν διάφορα τέτοια κριτήρια. Για να θεωρείται εγγυημένη η ορθότητα των προγραμμάτων το κάθε κριτήριο πρέπει να ικανοποιεί την συνθήκη για αξιοπιστία (reliability). Με τον όρο αξιοπιστία εννοούμε ότι ένα κριτήριο ελέγχου πρέπει να παράγει συνεπή αποτελέσματα ελέγχου, δηλαδή αν το πρόγραμμα εξετάσθηκε κάτω από μία ομάδα test cases που ικανοποιούν ένα κριτήριο, τότε το πρόγραμμα εξετάσθηκε κάτω από όλες τις ομάδες test cases, που ικανοποιούν το συγκεκριμένο κριτήριο. Έχει καθοριστεί ακόμα η συνθήκη της ισχύς (validity). Με τον όρο ισχύς εννοούμε ότι για κάθε λάθος σε ένα πρόγραμμα υπάρχει μια ομάδα από test cases που να ικανοποιούν το κριτήριο και να εντοπίζει το λάθος. Στη πορεία όμως παρατηρήθηκε ότι αυτές οι δύο συνθήκες δεν μπορούν να συνυπάρξουν [21].

Γενικά για τα κριτήρια ελέγχου ισχύουν τα ακόλουθα:

- Ένα επαρκή κριτήριο ελέγχου αποτελεί το κανόνα για τερματισμό του ελέγχου, δηλαδή προσδιορίζει αν έγινε αρκετός έλεγχος και μπορεί να σταματήσει.
- Ένα επαρκή κριτήριο ελέγχου παρέχει εκτίμηση/ μέτρηση της ποιότητας ελέγχου.

Έτσι ένα κριτήριο ελέγχου αποτελεί το κύριο συστατικό κάθε μεθόδου ελέγχου, αφού δίνει την προδιαγραφή για την επιλογή κάθε test case (η ακόμη δίνει την δυνατότητα παραγωγής test case με διάφορους αλγόριθμούς), και μετρά την ποιότητα κάθε ομάδας test cases.

Τα κριτήρια ελέγχου, σύμφωνα με την πηγή της πληροφορίας που χρησιμοποιήθηκε για να καθοριστούν οι απαιτήσεις ελέγχου, μπορούν να διαχωριστούν στις ακόλουθες ομάδες,:

- Specification based: προσδιορίζει τα κριτήρια ελέγχου σύμφωνα με τις αναγνωρίσιμες λειτουργίες (χαρακτηριστικά) των απαιτήσεων του λογισμικού.
- Program based: προσδιορίζει τον απαιτούμενο έλεγχο σύμφωνα με το πρόγραμμα που προορίζεται για έλεγχο.
- Combined specification: συνδυασμός των πιο πάνω.
- Interface based criteria: προσδιορίζει τα κριτήρια ελέγχου σύμφωνα με τον τύπο και το εύρος τιμών των εισόδων του λογισμικού, χωρίς να χρειάζεται αναφορά στις εσωτερικές λειτουργίες του λογισμικού.

Τα specification based και interface based criteria ανήκουν στην κατηγορία του black-box testing ενώ τα άλλα δύο στην κατηγορία του white-box testing.

Επίσης τα κριτήρια ελέγχου μπορούν να διαχωριστούν στις ακόλουθες ομάδες, σύμφωνα με την βασική προσέγγιση ελέγχου:

- Structural testing: Προσδιορίζει τα κριτήρια ελέγχου σύμφωνα με την κάλυψη συγκεκριμένων ομάδων στοιχείων της δομής του προγράμματος ή των προδιαγραφών του.
- Fault-base testing: Επικεντρώνεται στον εντοπισμό λαθών στο λογισμικό και εξαρτάται μερικώς από την ικανότητα εντοπισμού λαθών των test cases.
- Error based testing: Απαιτεί από διάφορα test cases να ελέγξουν το πρόγραμμα σε συγκεκριμένα σημεία που κατά την γνώση μας το πρόγραμμα διαφέρει από τις προδιαγραφές του.

Οι πιο πάνω τρόποι διαχωρισμού μπορούν να θεωρηθούν ως δύο διαστάσεις στον χώρο των κριτηρίων επάρκειας ελέγχου. Ένα κριτήριο ελέγχου μπορεί να ταξινομηθεί σύμφωνα με τις δυο πιο πάνω διακρίσεις. Για παράδειγμα ένα κριτήριο κάλυψης μπορεί να ανήκει στο structural testing και στο program-based criteria (π.χ. Control flow Criteria, Data flow Criteria).

Οι πιο βασικές ομάδες που βασίζονται στο program – based και structural testing είναι το κριτήριο ροής ελέγχου (Control Flow Criteria) και το κριτήριο ροής δεδομένων (Data Flow Criteria). Αυτές οι δύο ομάδες μπορούν να συνδυασθούν και να επεκταθούν ώστε να μας δώσουν το dependence coverage criteria.

2.10. Κριτήρια κάλυψης ροής ελέγχου (Control flow coverage)

Η κάλυψη ροής ελέγχου είναι η διαδικασία που:

- Βρίσκει περιοχές του προγράμματος (μονοπάτια) που δεν εξετάστηκαν από ένα σύνολο από test cases.
- Δημιουργεί επιπρόσθετα test cases για να αυξήσει την κάλυψη (coverage).
- Ορίζει ένα ποσοτικό μέτρο της κάλυψης του κώδικα, το οποίο είναι ένα έμμεσο μέτρο ποιότητας.

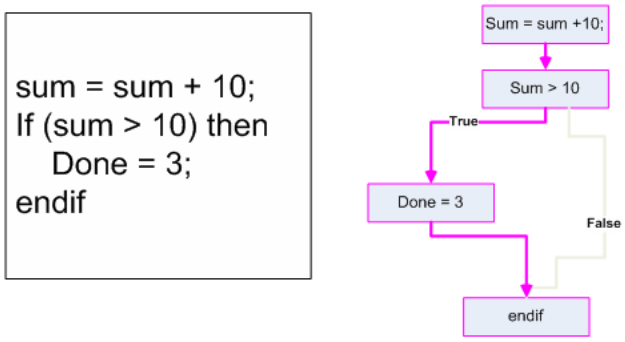
Το πέρασμα κάθε μονοπατιού εξαρτάται από πολλούς παράγοντες, όπως την επιλογή των αρχικών τιμών ή των τιμών εισόδων που ζητήθηκαν κατά την διάρκεια εκτέλεσης του προγράμματος. Πολύ σπάνια μπορεί κάποιος να ελέγξει όλα τα μονοπάτια ενός προγράμματος. Παρόλα αυτά η εκτέλεση ενός μονοπατιού δεν μπορεί να εξαρτηθεί τις περισσότερες φορές μόνο από τις τιμές εισόδων λόγω του ταυτοχρονισμού και του μη ντετερμινισμού των εφαρμογών [13].

Το unit testing συνήθως γίνεται με βάση κάποια κριτήρια κάλυψης (coverage criteria) τα οποία μπορούν να ομαδοποιήσουν κάποιες εκτελέσεις και να δώσουν την δυνατότητα σε κάποιο που ελέγχει το πρόγραμμα να εφαρμόσει μία εκτέλεση από το κάθε σύνολο. Ακολουθούν περιγραφές των κυριότερων κριτηρίων κάλυψης.

2.10.1. Κάλυψη εντολής (Statement Coverage)

Το statement coverage κριτήριο λέει ότι κάθε εκτελέσιμη εντολή (statement) στο πρόγραμμα πρέπει να εκτελεστεί τουλάχιστον μία φορά. Αυτό που βασικά απαιτείται από τους ελεγκτές προγραμμάτων, είναι να βρουν δοκιμές ελέγχου (test cases) ώστε να εκτελεστούν όλες οι εντολές στο πρόγραμμα. Μία ομάδα από test cases (test suit) που ικανοποιεί το κριτήριο αυτό είναι επαρκές κατά το κριτήριο του statement coverage. Μερικές φορές υπολογίζεται το ποσοστό εκτέλεσης των εντολών για να μετρηθεί η επάρκεια ελέγχου σύμφωνα με το κριτήριο.

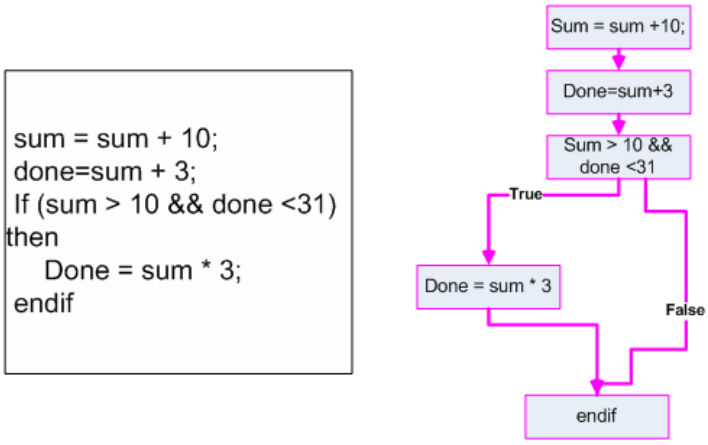
Το μειονέκτημα είναι ότι πιθανόν να υπάρχουν μονοπάτια τα οποία δεν εκτελούνται. Η κάλυψη δήλωσης (statement coverage) μπορεί να γίνει αποδεκτή ως ελάχιστη απαίτηση ελέγχου. Μπορεί να χρησιμοποιηθεί στο επίπεδο ενότητας με λιγότερο από 5.000 γραμμές κώδικα. Για να καταστεί αποτελεσματική αυτή η δοκιμή, η κάλυψη 100% είναι απαραίτητη. Η κάλυψη δήλωσης έχει περίπου το 50% της αποτελεσματικότητας του branch coverage.



Σχήμα 2.6: Συνθήκη κάλυψης εντολής

2.10.2. Κάλυψη Ακμής (Edge Coverage)

Κάθε εκτελέσιμη πλευρά (edge) του διαγράμματος ροής (flowchart) πρέπει να καλύπτεται τουλάχιστον από ένα test case.



Σχήμα 2.7: Συνθήκη κάλυψης ακμής

Το μειονέκτημα είναι ότι όταν έχουμε σύνθετα κατηγορήματα σε κόμβο απόφασης, δεν ελέγχουμε ξεχωριστά την κάθε συνθήκη αλλά το σύνολο της συνθήκης.

2.10.3. Κάλυψη Συνθήκης (Condition Coverage)

Αυτό το κριτήριο αναγνωρίζει ότι κάθε κατηγόρημα απόφασης (decision predicate), στην πραγματικότητα μπορεί να είναι σύνθετος συνδυασμός πολλών απλών κατηγορημάτων (simple first order formulas). Στα condition coverage criteria πρέπει κάθε εκτελέσιμο κατηγόρημα (condition) να αποτιμάται από κάποιο test case ως αληθές (true) και από κάποιο άλλο ως ψευδές (false).

Το μειονέκτημα του κριτηρίου είναι ότι μπορεί στους συνδυασμούς που θα δημιουργηθούν να έχουμε false ^ true ή true ^ false με επακόλουθο το τελικό

αποτέλεσμα να οδηγείται στη μη εκτέλεση κάποιας πλευράς του flowchart. Η κάλυψη συνθήκης (branch/ condition coverage) είναι η αποτελεσματικότερη για τη δοκιμή κάλυψης ροής ελέγχου που ολοκληρώνεται στο επίπεδο ενότητας. Το απαραίτητο επίπεδο κάλυψης είναι 85%.

2.10.4. Κάλυψη Ακμής/ Συνθήκης (Edge/Condition Coverage)

Αυτό το κριτήριο απαιτεί να καλυφθούν κάθε πλευρά και συνθήκη του flowchart.

2.10.5. Κάλυψη Πολλαπλής Συνθήκης (Multiple Condition Coverage)

Αυτό το κριτήριο απαιτεί όπως, κατά την εκτέλεση ενός προγράμματος, να καλυφθούν όλοι οι Boolean συνδυασμοί των συνθηκών οι οποίοι μπορούν να συναντηθούν σε κάθε κατηγόρημα απόφασης από κάποια test case.
Το Multiple Condition είναι πιο εξαντλητικό κριτήριο από τα προηγούμενα κριτήρια αφού απαιτεί περισσότερα test case για να καλυφθούν όλοι οι συνδυασμοί.

2.10.6. Κριτήριο Κάλυψης Μονοπατιού (Path Coverage Criteria)

Αυτό το κριτήριο απαιτεί να καλυφθούν όλα τα εκτελέσιμα μονοπάτια σε κάποια test case. Δυστυχώς ο αριθμός των μονοπατιών σε ένα πρόγραμμα μπορεί να τεράστιος. Δεν θα μπορούσαμε να απαιτήσουμε κάλυψη αυτού του κριτήριου σε ένα επαναληπτικό βρόγχο (Loop) για τον λόγο ότι μπορεί προκύψουν απεριόριστα και μη πρακτικά μονοπάτια.

Κάποια μονοπάτια μπορεί να μην έχουν την δυνατότητα να εκτελεστούν ποτέ όπως και κάποια κατηγορήματα απόφασης που μπορεί να μην έχουν την δυνατότητα να υπολογιστούν σε μια δεδομένη τιμή αλήθειας (true ή false) και έτσι κάποιο σημείο κώδικα να είναι αδιαπέραστο (dead code).

Η ολοκλήρωση ενός συνόλου δοκιμών μονοπατιών μπορεί να πάρει από 8 έως 10 φορές περισσότερο χρόνο από την κάλυψη συνθηκών. Αυτό το είδος ελέγχου γίνεται για κρίσιμες ενότητες ενός κώδικα και περιορίζεται σε μερικά προγράμματα που μπορούν να χαρακτηριστούν σαν «ζωής και θανάτου» (ιατρικά συστήματα, ελεγκτές πραγματικού χρόνου).

2.10.7. Μειονεκτήματα κριτηρίων κάλυψης

Τα κριτήρια κάλυψης ροής ελέγχου (Control flow) έχουν διάφορους περιορισμούς. Συνήθως η κάλυψη είναι πολύ δύσκολη και σε μεγάλο εύρος. Δηλαδή είναι σχεδόν αδύνατον να καλυφθούν όλα τα μονοπάτια και μερικά λάθη του προγράμματος που βρίσκονται σε μονοπάτια που δεν καλύφθηκαν, δεν θα βρεθούν ποτέ.

Ακόμη ένα πρόβλημα είναι ότι ακόμα και αν εξετάσουμε όλα τα μονοπάτια σε ένα πρόγραμμα, είναι δύσκολο να εξετάσουμε κατά την διάρκεια ελέγχου ότι κάποιο μονοπάτι κάνει ακριβώς ότι έπρεπε να κάνει σύμφωνα με τις προδιαγραφές του συστήματος.

Τέλος αν έχουμε δυο εκτελέσεις προγραμμάτων που εκτελούνται ακριβώς το ίδιο και παρουσιάζουν λάθος στο ίδιο σημείο τότε, παίρνουμε μόνο ένα test case για το ένα και αγνοούμε το άλλο, αφού και η συμπεριφορά και των δυο προγραμμάτων μπορεί να ελεγχθεί μόνο με ένα test case.

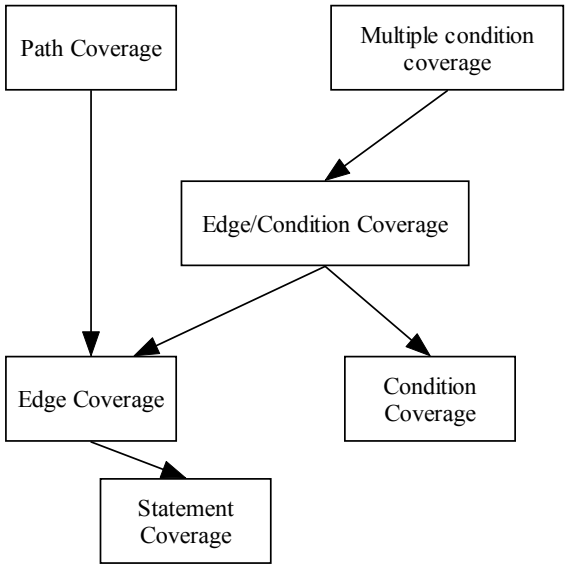
2.10.8. Σύγκριση των κριτηρίων κάλυψης

Τα διάφορα κριτήρια κάλυψης συσχετίζονται και θα μπορούσαμε να πούμε ότι ένα κριτήριο είναι υποσύνολο του άλλου, και αν καλύψουμε το κριτήριο που είναι υπέρ-σύνολο (το οποίο σίγουρα χρειάζεται επιπρόσθετη δουλειά) τότε αυτόματα καλύπτουμε και τα υποδεέστερα του.

Έτσι μπορούμε να συγκρίνουμε συσχετικές δυνάμεις αφού ένα δυνατότερο κριτήριο εμπεριέχει ένα πιο αδύνατο κριτήριο. Παρακάτω δείχνω τις συσχετίσεις μεταξύ των κριτηρίων:

- Το edge coverage εμπεριέχει το statement coverage αφού εξετάζοντας κάθε ακμή οδηγεί στην εξέταση κάθε statement.
- Το Edge/Condition Coverage εμπεριέχει το edge coverage και το condition coverage.
- Το path coverage εμπεριέχει το edge coverage.

Ακαδημαϊκά μιλώντας το δυνατότερο κριτήριο υπάγει το πιο αδύνατο. Οι coverage μετρικές δεν μπορούν να συγκριθούν ποσοτικά.



Σχήμα 2.8: Ιεραρχική κατάταξη των κριτηρίων κάλυψης

Πολύ εύκολα μπορούμε να παρατηρήσουμε και από το σχήμα ότι το edge coverage εντάσσει το statement coverage ενώ το edge/condition coverage εντάσσει το edge coverage (και αυτόματα το statement coverage) και το condition coverage.

2.11. Τι είναι οι γράφοι ροής δεδομένων

Οι γράφοι ροής δεδομένων είναι μια γραφική αναπαράσταση της "ροής" των δεδομένων διαμέσου ενός πληροφοριακού συστήματος (σύστημα το οποίο επεξεργάζεται δεδομένα και πληροφορίες). Ένας γράφος ροής δεδομένων χρησιμοποιείται για την απεικόνιση της επεξεργασίας των δεδομένων (δομημένη σχεδίαση).

Ανακαλύφθηκαν από τον Larry Constantine, τον αρχικό εφευρέτη του δομημένου σχεδιασμού (structured design). Με τους γράφους ροής δεδομένων, οι χρήστες των συστημάτων είναι ικανοί να δουν πως το σύστημα λειτουργεί, τι μπορεί να πραγματοποιήσει και πως είναι αναπτυγμένο.

Οι κόμβοι ενός Γράφου Ροής Δεδομένων (data flow graph – DFG) αντιπροσωπεύουν διαδικασίες και οι ακμές αντιπροσωπεύουν τα μονοπάτια που ακολουθούν τα δεδομένα. Μπορούν να χρησιμοποιηθούν για έλεγχο, αφού η μόνη διαφορά από τους γράφους ροής ελέγχου είναι ότι εξετάζουν το κύκλο ζωής μια μεταβλητής. Τα κριτήρια κάλυψης των γράφων ροής δεδομένων βασίζονται στο πότε και πώς ορίζονται (ή δημιουργούνται) οι μεταβλητές και πώς αυτές χρησιμοποιούνται στο πρόγραμμα. Έτσι, τα κριτήρια αυτά επικεντρώνονται σε λανθασμένη χρήση μεταβλητών λόγο λάθους στον κώδικα [14]. Τέτοια λάθη γίνονται συχνά από προγραμματιστές όπως για παράδειγμα η χρήση μιας μεταβλητής πριν αυτή οριστεί ή πριν της δοθεί τιμή.

Με την ανάλυση με χρήση γράφων ροής δεδομένων μπορεί κάποιος να ελέγξει τα μονοπάτια από τον ορισμό μια μεταβλητής μέχρι την χρησιμοποίηση της και το πώς γίνεται η ροή μιας τιμής σε αυτά τα μονοπάτια.

Αναλυτική παρουσίαση των γράφων ροής δεδομένων και πιο αυστηροί ορισμοί θα δοθούν στο επόμενο κεφάλαιο. Θα παρουσιαστούν επίσης τα διάφορα κριτήρια που έχουν προταθεί για δυναμικό έλεγχο κώδικα λογισμικού.

2.12. Συνθετικότητα (Compositionality)

Συνήθως μεγάλα λογισμικά αναπτύσσονται από διαφορετικές ομάδες ανθρώπων, όπου η κάθε μια είναι υπεύθυνη για ένα κομμάτι του κώδικα. Η ίδια ακριβώς τεχνική ακολουθείται και στην διαδικασία ελέγχου του λογισμικού. Δηλαδή, διάφορα κομμάτια (modules) του προγράμματος ελέγχονται από διαφορετικές ομάδες ανθρώπων. Αυτή η τεχνική έχει ως αποτέλεσμα να ανακαλύπτονται λάθη σε μικρότερα κομμάτια κώδικα και να επιτυγχάνεται καλύτερη διαχείριση της πολυπλοκότητας του συστήματος.

Υποθέτουμε ότι έχουμε το module A που καλεί το module B. Για να ελέγξουμε το A πριν είναι διαθέσιμο το B, γράφουμε απλές και μικρές εκδόσεις του module B, τα λεγόμενα stubs. Stubs δηλαδή είναι κομμάτι κώδικα που εξομοιώνει μια καλούμενη συνάρτηση. Για να ελέγξουμε το B πριν είναι διαθέσιμο το A, τότε γράφουμε ενδεικτικό κώδικα για το module A, το λεγόμενο driver. Driver δηλαδή είναι κομμάτι κώδικα που εξομοιώνει μια συνάρτηση που θα καλούσε μια άλλη.

Πώς όμως, μπορεί να γίνει ο έλεγχος; Υπάρχουν δύο τρόποι με τους οποίους μπορεί να επιτευχθεί σωστά ο έλεγχος ενός σύνθετου λογισμικού.

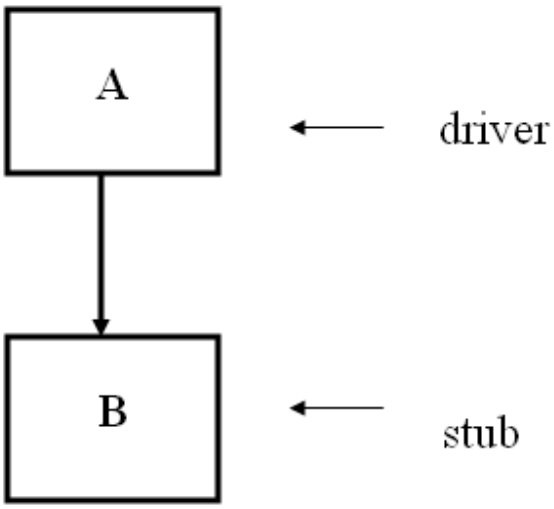
2.12.1. Έλεγχος Bottom-up

Αυτός ο τρόπος ελέγχου, ξεκινά από τα χαμηλότερα επίπεδα στην ιεραρχία (χαμηλότερα modules). Έτσι με αυτή την τεχνική τα λειτουργικά modules ελέγχονται πολλές φορές με τα drivers που γράφονται και όχι μέσω modules που μπορεί να περιέχουν λάθη. Με αυτό τον τρόπο ελέγχου, βασικά λάθη στη σχεδίαση φαίνονται σχετικά αργά.

2.12.2. Έλεγχος Top-down

Αυτή ο τρόπος ελέγχου ξεκινά από το ψηλότερο επίπεδο της ιεραρχίας (ψηλότερο module). Τα υπόλοιπα modules υλοποιούνται σαν stub και στην πορεία επεκτείνονται. Με αυτό τον τρόπο απομονώνονται λάθη και τα stubs μετά αξιοποιούνται,

αφού επεκτείνονται στα αντίστοιχα πλήρης modules. Ακόμη, βασικά λάθη στη σχεδίαση φαίνονται νωρίς, αλλά όμως λειτουργικά modules δεν ελέγχονται πολλές φορές.



Σχήμα 2.9: Έλεγχος Top-down και Bottom-up

Κεφάλαιο 3

Γράφος ροής Δεδομένων

3.1. Εισαγωγικά.....41
3.2. Βασικές έννοιες.....42
3.3. Σημαντικότερες μέθοδοι πλοήγησης σε γράφο 48
3.4. Το κριτήριο του Ntafos.....50
3.5. Τα κριτήρια των Rapps και Weyuker.....56
3.6. Τα κριτήρια Laski και Korel 63
3.7. Σύγκριση των κριτηρίων.....67

3.1. Εισαγωγικά

Στα μαθηματικά αλλά και στην επιστήμη των ηλεκτρονικών υπολογιστών, η θεωρία γράφων είναι η μελέτη των γράφων η οποία αναφέρεται σε μια συλλογή από κόμβους και ακμές. Πιο συγκεκριμένα οι γράφοι ροής δεδομένων (data flow graph – DFG) είναι μια γραφική αναπαράσταση, όπου οι κόμβοι αντιπροσωπεύουν διαδικασίες και οι ακμές αντιπροσωπεύουν τα μονοπάτια που ακολουθούν τα δεδομένα (φαίνεται η ροή των δεδομένων). Τα κριτήρια κάλυψης των γράφων ροής δεδομένων βασίζονται στο πότε και πώς ορίζονται (ή δημιουργούνται) οι μεταβλητές και πώς αυτές χρησιμοποιούνται στο πρόγραμμα. Σκοπός της ανάλυσης με χρήση DFG είναι να ελέγξει τα μονοπάτια από τον ορισμό μια μεταβλητής μέχρι την χρησιμοποίηση της και το πώς γίνεται η ροή μιας τιμής σε αυτά τα μονοπάτια.

Για τους γράφους ροής ελέγχου σε αντίθεση με τους DFG, υπάρχουν σαφή κριτήρια, αποδεκτά από ολόκληρη την ακαδημαϊκή και ερευνητική κοινότητα. Για τους γράφους ροής δεδομένων υπάρχουν τα κριτήρια τα οποία έχουν προταθεί από τρεις διαφορετικούς συγγραφείς ή ομάδες, τα οποία χαίρουν ιδιαίτερης εκτίμησης, και πάνω στα οποία έχουν στηριχτεί σχεδόν όλες οι προσπάθειες ανάλυσης με βάση τους DFG.

Σε αυτό το κεφάλαιο θα παρουσιαστούν οι τρεις διαφορετικές προτάσεις και θα ακολουθήσει σύγκριση τους. Η σύγκριση και ιεράρχηση των κριτηρίων αυτών αποτέλεσε από μόνο του αντικείμενο έρευνας. Της παρουσίασης των κριτηρίων θα προηγηθεί παρουσίαση χρήσιμων για την συνέχεια ορισμών καθώς και μερικές ανωμαλίες πηγαίου κώδικα οι οποίες μπορούν να ανιχνευθούν με στατική ανάλυση του γράφου ροής δεδομένων.

3.2. Βασικές έννοιες

Πριν προχωρήσω στον ορισμό των κριτηρίων πρέπει να εισάγω κάποιες βασικές έννοιες, για τους γράφους ροής δεδομένων, ώστε να μπορέσει ο αναγνώστης να κατανοήσει αρκετά πράγματα που ακολουθούν στην εργασία αυτή.

3.2.1. Ορισμός του γράφου ροής δεδομένων

Καταρχάς πριν δώσω τον ορισμό να εξηγήσω κάποιες έννοιες που χρησιμοποιούνται στον ορισμό:

- Ένας γράφος ονομάζεται **κατευθυνόμενος γράφος** (directed graph, digraph) αν κάθε μια από τις ακμές του είναι προσανατολισμένη προς μια κατεύθυνση.
- Ένας γράφος ονομάζεται **συνεκτικός γράφος** (connected graph) αν δεν υπάρχει κόμβος που να μην είναι με κάποιο τρόπο ενωμένος με το υπόλοιπο σύνολο.
- Ένας γράφος συμβολίζεται και ως G = (N, A), όπου N ο αριθμός όλων τον κόμβων (vertices) και όπου A ο αριθμός όλων των ακμών (edges).

Τώρα μπορώ να προχωρήσω στον πιο κάτω ορισμό που δόθηκε στο [15] και ισχύει σε ότι ακολουθεί στην παρούσα εργασία.

> Ο DFG G = (N, A), ενός προγράμματος είναι ένας κατευθυνόμενος, συνεκτικός γράφος που έχει ένα μοναδικό σημείο εισόδου και ένα μοναδικό σημείο εξόδου. Ένας κόμβος n που ανήκει στο σύνολο N, μπορεί να αντιπροσωπεύει μία εντολή ή ένα μπλοκ από εντολές, όπου το κάθε μπλοκ έχει ένα μοναδικό σημείο εισόδου και ένα σημείο εξόδου.

Μια ανωμαλία ροής δεδομένων μπορεί να χαρακτηριστεί ή μη λογική ή αχρείαστη αλλαγή στην κατάσταση ενός «αντικειμένου δεδομένων» (για παράδειγμα στην τιμή μια μεταβλητής). Οι ανωμαλίες αυτές δείχνουν σημεία στα οποία πιθανόν να υπάρχει σφάλμα στο πρόγραμμα. Οι πιο κάτω έννοιες θεωρούνται χρήσιμες για την παρακολούθηση και κατανόηση των όσων θα ακολουθήσουν σε αυτή την εργασία.

3.2.2. Τι είναι στατική ανάλυση

Ας υποθέσουμε ότι το υπό έλεγχο πρόγραμμα έχει μια μεταβλητή *X*. Αυτή η μεταβλητή μπορεί να υποστεί αλλαγές ή να εφαρμοστούν σε αυτή οι πιο κάτω ενέργειες:

1. **Ορισμός μεταβλητής (define – d)**: Η μεταβλητή X ορίζεται όταν της δοθεί τιμή.

2. **Χρήση της μεταβλητής (use - u)**: Η τιμή της μεταβλητής Χ χρησιμοποιείται. Μπορεί να χρησιμοποιηθεί είτε σαν μέρος ενός υπολογισμού δηλαδή στο δεξιό μέλος του ίσον (right hand side) είτε ως μέρος υπολογισμού μιας συνθήκης αληθείας (predicate or condition). Οι δυο αυτές διαφορετικές χρήσεις συμβολίζονται ως c-use και p-use αντίστοιχα.
3. **Καταστροφή της μεταβλητής (undefined - n)**: Η μεταβλητή χάνει την τιμή της ή η τιμή της γίνεται άγνωστη ή αβέβαιη.

Οι ανωμαλίες που μπορεί να προκύψουν από ένα πρόγραμμα κατά την διάρκεια χρήσης μεταβλητών προκαλούνται από την σειρά κατά την οποία το πρόγραμμα ορίζει, χρησιμοποιεί ή καταστρέφει τις μεταβλητές αυτές. Με βάση αυτό τον ορισμό μπορούν να διακριθούν τέσσερα διαφορετικά είδη ανωμαλιών.

1. Ανωμαλία n – u: Προσπάθεια χρήσης μιας μεταβλητής που έχει καταστραφεί.
2. Ανωμαλία d – n: Μια μεταβλητή που έχει οριστεί δεν χρησιμοποιείται καθόλου πριν καταστραφεί ή αλλάξει η τιμή της.
3. Ανωμαλία d – d: Μια μεταβλητή ορίζεται 2 φορές, προκαλώντας προβλήματα στην εμβέλεια του πρώτου ορισμού της.
4. Ανωμαλία n – n: Προσπάθεια καταστροφής της ίδιας μεταβλητής δύο φορές.

Με βάση τους πιο πάνω ορισμούς είναι ξεκάθαρο ότι η στατική ανάλυση με DFG μπορεί να ανιχνεύσει λάθη σε ένα πρόγραμμα που προκαλούνται κατά την πληκτρολόγηση, από μη αρχικοποίηση μεταβλητών και από χρήση λάθους μεταβλητής σε συγκεκριμένο σημείο.

3.2.3. Ορισμός και χρήση μεταβλητής

Τα κριτήρια που θα παρουσιαστούν στην συνέχεια, είναι βασισμένα σε μια εξερεύνηση του τρόπου με τον οποίο οι τιμές συνδέονται με τις μεταβλητές, και πώς αυτές οι συνδέσεις μπορούν να έχουν επιπτώσεις στην εκτέλεση του προγράμματος. Η ανάλυση εστιάζεται στις εμφανίσεις μεταβλητών μέσα στο πρόγραμμα ενώ οι ακριβείς συναρτήσεις και υπολογισμοί μιας συνθήκης δεν παίζουν κανένα ρόλο. Κάθε εμφάνιση μιας μεταβλητής μπορεί να ενταχθεί σε μια από τρεις τάξεις:

1. Ως περιστατικό ορισμού μεταβλητής (definition - **def**),
2. Ως εμφάνιση και χρήση σε ένα υπολογισμό (computational use ή **c-use**),
3. Ως χρήση για υπολογισμό μιας παράστασης αληθείας (predicate use ή **p-use**).

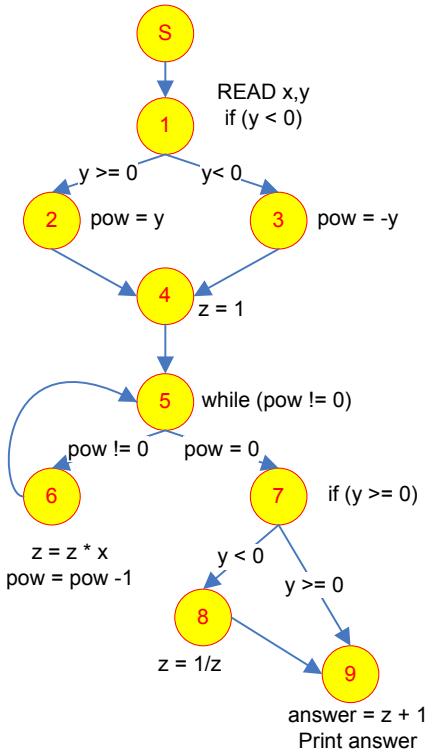
Στο εξής θα γίνεται αναφορά στους πιο πάνω πιθανούς τρόπου χρήσης ως def, c-use και p-use αντίστοιχα.

Μερικά παραδείγματα των ορισμών αυτών ακολουθούν:

- Η εντολή ανάθεσης $y = f(x_1,...x_n)$ περιέχει c-use των $x_1,...x_n$ ακολουθούμενη από def του y.
- Η εντολή εισόδου "*read* $x_1,...x_n$" περιέχει def των $x_1,...x_n$.
- Η εντολή "$if\ p(x_1,...x_n)\ then\ goto\ m$" περιέχει p-use των $x_1,...x_n$.

Σε ένα γράφο ροής δεδομένων, οι κόμβοι περιέχουν είτε def είτε c-use μιας ή περισσοτέρων μεταβλητών. Επειδή η τιμή μιας μεταβλητής σε μια συνθήκη ελέγχου (δηλαδή το p-use), επηρεάζει την ροή εκτέλεσης εντολών, συσχετίζουμε το p-use με ακμή αντί με κόμβο. Αν μια εντολή στον κόμβο i είναι συνθήκη if-then-else με συνθήκη ελέγχου το (x > 0) που μπορεί να οδηγήσει είτε στον κόμβο j είτε στον κόμβο k, τότε λέμε ότι οι ακμές (i, j) και (i, k) περιέχουν p-use του x. Τα πιο πάνω παρουσιάζονται σε ένα παράδειγμα για περισσότερη κατανόηση του αναγνώστη. Χρησιμοποιείται μια συμβολική γλώσσα χωρίς καμία απώλεια της γενικότητας.(Πιο κάτω έχουμε ένα module στα αριστερά και το γράφο ροής δεδομένων του στα δεξιά.)

```
 Εντολή                    Κόμβος
 1. start                    0
 2. read x,y                 1
 3. if y<0 -> goto 6         1
 4. pow := y                 2
 5. goto 7                   -
 6. pow := -y                3
 7. z:= 1                    4
 8. if pow=0 -> goto 12      5
 9. z := z*x                 6
10. pow := pow-1             6
11. goto 8                   -
12. if y>=0 -> goto 14       7
13. z := 1/z                 8
14. answer := z+1            9
15. print answer             9
16. stop                     9
```



Σχήμα 3.1: Γράφος Ροής Δεδομένων του Προγράμματος

Ακολουθώντας του ορισμούς για def, c-use και p-use, για το πιο πάνω πρόγραμμα και τον γράφο ροής δεδομένων του τότε για κάθε κόμβο και για κάθε ακμή έχουμε τα εξής: (οι ακμές οι οποίες δεν περιέχουν p-use δεν αναφέρονται στον πιο κάτω πίνακα)

C – USES και DEFINITIONS			**P – USES**	
node	**c-use**	**definition**	**edge**	**p-use**
1	None	x, y	(1, 2)	y
2	y	pow	(1, 3)	y
3	y	pow	(5, 6)	pow
4	None	z	(5, 7)	pow
5	None	None	(7, 8)	y
6	x, z, pow	z, pow	(7, 9)	y
7	None	None		
8	z	z		
9	z	answer		

Πίνακας 3.1: C-uses και p-uses για τις μεταβλητές του γράφου-σχήματος 3.1

Έστω ότι το X είναι μια μεταβλητή που εμφανίζεται σε ένα πρόγραμμα. Λέμε ότι ένα μονοπάτι $(i, n_1, \ldots n_m, j), m \geq 0$, το οποίο δεν περιέχει κανένα def του X στους κόμβους $i, n_1, \ldots n_m, j$ ονομάζεται μονοπάτι καθαρό από ορισμούς (**def-clear path**) όσον αφορά το X από τον κόμβο i στον κόμβο j. Ένα μονοπάτι $(i, n_1, \ldots n_m, j, k), m \geq 0$, που δεν περιέχει κανένα def του X στους κόμβους $i, n_1, \ldots n_m, j$, ονομάζεται **def-clear** μονοπάτι όσον αφορά το X από τον κόμβο i στην ακμή (j, k). Μια ακμή (i, j) είναι από μόνη της ένα **def-clear** μονοπάτι όσον αφορά το X από τον κόμβο i μέχρι την ακμή (i, j) γιατί αποκλείεται ποτέ να έχουμε definition σε μια ακμή.

Ένα def μιας μεταβλητής X στον κόμβο i είναι ένα σφαιρικό def (global def) εάν και μόνο εάν είναι το τελευταίο def του X που εμφανίζεται στο μπλοκ που συνδέεται με τον κόμβο i και υπάρχει ένα def-clear μονοπάτι όσο αφορά το X από το i είτε στον κάθε κόμβο που περιέχει μια c-use του X είτε σε μια ακμή που περιέχει ένα p-use του X. Ένα def μιας μεταβλητής X στον κόμβο i που δεν είναι σφαιρικό def (global def) είναι ένα τοπικό def (local def) εάν και μόνο εάν υπάρχει ένα τοπικό c-use του X στον κόμβο i που δεν ακολουθεί αυτό το def, και κανένα άλλο def του X δεν εμφανίζεται μεταξύ του def και του τοπικού c-use. Το def της μεταβλητής answer στον κόμβο 9 του πιο πάνω

σχήματος είναι τοπική. Οποιοδήποτε def που δεν είναι ούτε σφαιρικό ούτε τοπικό δεν θα χρησιμοποιηθεί ποτέ και το πρόγραμμα πρέπει να εξεταστεί για πιθανό λάθος.

Δημιουργούμε το γράφο def/use από τον control flow graph με το να συσχετίζουμε κάθε κόμβο i με τα δύο σύνολα def και c-use αλλά και την κάθε ακμή (i, j) με το σύνολο p-use. Ονομάζουμε **def(i)** το σύνολο μεταβλητών για τις οποίες ο κόμβος i περιέχει ένα global def , **c-use(i)** το σύνολο μεταβλητών για το οποίο ο κόμβος i περιέχει ένα global c-use και **p-use(i,j)** το σύνολο από μεταβλητές για τις οποίες η ακμή (i,j) περιέχει ένα p-use. Μια ακμή (i, j) για την οποία το σύνολο p-use(i, j) δεν είναι άδειο καλείται επονομαζόμενη (**labeled**) ακμή εάν όμως p-use(i, j) = 0 τότε καλείται μη επονομαζόμενη (**unlabelled**) ακμή. Η ακμή που είναι η μοναδική εξερχόμενη από έναν κόμβο είναι πάντα unlabelled, ενώ οι ακμές που είναι μέρος ζευγαριού (π.χ. μετά από ένα if) εξερχόμενων ακμών είναι πάντα labelled.

Συνεχίζοντας με το πιο πάνω παράδειγμα και μετά και την εισαγωγή των τελευταίων ορισμών, παρατηρούμε ότι η μεταβλητή answer έχει μόνο ένα local def και ένα local c-use. Οι ακμές (2,4), (3,4), (4,5), (6,5) και (8,9) είναι unlabelled.

Καθορίζουμε τώρα διάφορα σύνολα που απαιτούνται κυρίως για τα κριτήρια που θα χρησιμοποιηθούν στην συνέχεια της εργασίας με βάση αυτά που πρότειναν οι S. Rapps και E. J. Weyuker [16], [17], [18]. Ας είναι το i ένας οποιοσδήποτε κόμβος, και το X οποιαδήποτε μεταβλητή έτσι ώστε $X \in def(i)$. Τότε το dcu(X, i) είναι το σύνολο όλων των κόμβων j έτσι ώστε $X \in c-use(j)$ και για το οποίο υπάρχει ένα def-clear μονοπάτι όσο αφορά το X από το i στο j. Το dpu(X, i) είναι το σύνολο όλων των ακμών (j, k) έτσι ώστε $X \in p-use(j,k)$ και για το οποίο υπάρχει ένα def-clear μονοπάτι όσο αφορά το X από το i στην (j, k). Τα σύνολα dcu και dpu για το Σχήμα 3.1 είναι:

Μεταβλητή	Κόμβος	DCU	DPU
x	1	{6}	∅
y	1	{2, 3}	{(1, 2), (1, 3), (7, 8), (7, 9)}
pow	2	{6}	{(5, 6), (5, 7)}
pow	3	{6}	{(5, 6), (5, 7)}
z	4	{6, 8, 9}	∅
z	6	{6, 8, 9}	∅
pow	6	{6}	{(5, 6), (5, 7)}
Z	8	{9}	∅

Πίνακας 3.2: DCU και DPU για τις μεταβλητές του γράφου-σχήματος 3.1

Στην συνέχεια του κεφαλαίου θα παρουσιάσουν οι προτάσεις από τρεις διαφορετικές ομάδες για πιθανά κριτήρια κάλυψης γράφων ροής δεδομένων. Όπου κρίνεται αναγκαίο εισάγονται νέοι ορισμοί για καλύτερη κατανόηση των κριτηρίων.

3.2.4. Υποθέσεις και παραδοχές

Υπάρχουν κάποιες παραδοχές για την ανάλυση των γράφων ροής δεδομένων. Οι παραδοχές αυτές έγιναν σε όλες τις προσπάθειες για δημιουργία των κριτηρίων που παρουσιάζω στην συνέχεια.

Οι υποθέσεις και παραδοχές είναι:

1. Δεν υπάρχουν ακμές της μορφής (n, n).
2. Υπάρχει το πολύ μια ακμή της μορφής (m, n) για κάθε m, n.
3. Ο κάθε γράφος είναι συνεκτικός και έχει ένα μοναδικό σημείο εισόδου και ένα μοναδικό σημείο εξόδου.
4. Κάθε βρόχος (loop) έχει ένα μοναδικό σημείο εισόδου και ένα μοναδικό σημείο εξόδου.
5. Σε μια παράσταση υπολογισμού τιμής αληθείας, δεν υπάρχουν ορισμοί μεταβλητών.

6. Κάθε ορισμός μια μεταβλητής, φτάνει σε μία τουλάχιστο χρήση.
7. Σε κάθε χρήση φτάνει ένας ορισμός.
8. Σε κάθε γράφο υπάρχει τουλάχιστον ένας ορισμός μεταβλητής.
9. Δεν υπάρχουν ορισμοί ή χρήσεις μεταβλητών στον αρχικό και τελικό κόμβο του γράφου.

3.3. Σημαντικότερες μέθοδοι πλοήγησης σε γράφο

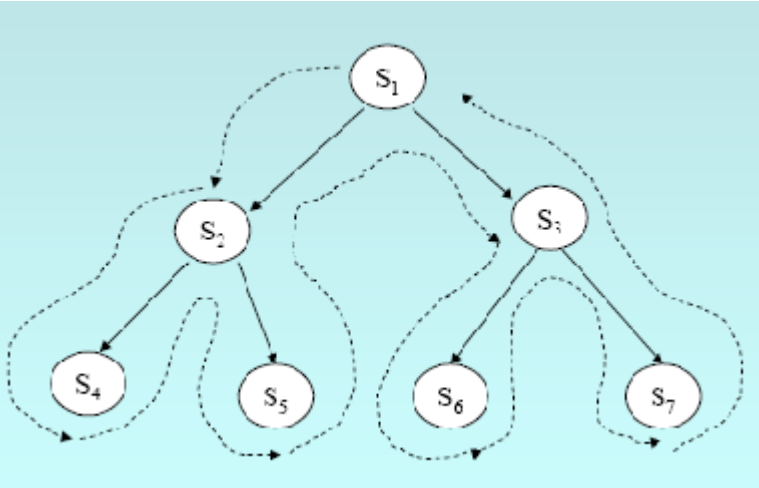
Οι μέθοδοι πλοήγησης σε γράφο επιλύουν το πρόβλημα επίσκεψης όλων των κόμβων ενός γράφου με ένα καθορισμένο τρόπο ο καθένας. Υπάρχουν πολλοί μέθοδοι πλοήγησης σε γράφους. Η κάθε μέθοδος έχει το δικό της κόστος και τα δικά της θετικά και αρνητικά. Παρακάτω επεξηγώ τις δύο γνωστότερες μεθόδους πλοήγησης σε γράφο.

3.3.1. Αναζήτηση σε βάθος (Depth first search)

Η αναζήτηση σε βάθος (Depth first search ή DFS) είναι ένας αλγόριθμος που διασχίζει ή εξερευνά ένα δέντρο ή ένα γράφο. Διαισθητικά, ξεκινά από την ρίζα (από κάποιο κόμβο που είναι η αρχή του γράφου) και εξερευνά όσο δυνατό πιο μακριά κατά μήκος κάθε διακλάδωσης (branch) πριν την οπισθοδρόμηση (backtracking).

Δεν ανακαλύπτει απαραιτήτως την βέλτιστη λύση όμως η απαίτηση της σε μνήμη δεν είναι υπερβολική. Προσπαθεί να εισχωρήσει γρήγορα στο χώρο αναζήτησης και μπορεί να φτάσει σε τελική κατάσταση με αποδοτικό τρόπο.

Τυπικά, ο αλγόριθμος DFS είναι μια «τυφλή μέθοδος αναζήτησης» που προχωρά επεκτείνοντας το πρώτο αριστερότερο παιδί-κόμβο του γράφου και κατά αυτό τον τρόπο προχωρά βαθύτερα και βαθύτερα μέχρι να βρεθεί ο κόμβος που γυρεύουμε ή μέχρι να φθάσει σε κόμβο που δεν έχει παιδιά. Τότε η αναζήτηση οπισθοδρομεί (backtracks), επιστρέφοντας στο πιο πρόσφατο κόμβο που δεν έχει τελειώσει την αναζήτηση του.



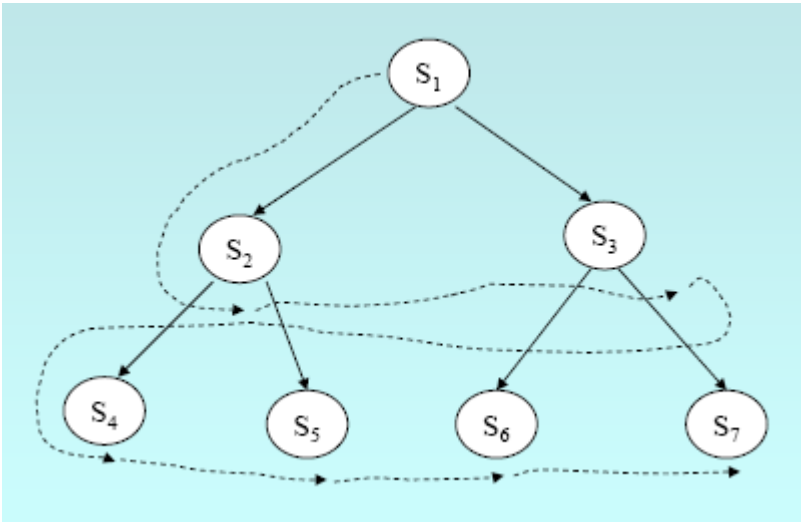
Σχήμα 3.2: Αναζήτηση σε βάθος

3.3.2. Αναζήτηση σε πλάτος (Breadth first search)

Η αναζήτηση σε πλάτος (Breadth first search ή BFS) είναι ένας αλγόριθμος εξερεύνησης γράφων που ξεκινά από την ρίζα (κόμβος που είναι η αρχή του γράφου) και εξετάζει όλους τους γειτονικούς κόμβους. Μετά από όλους αυτούς τους γειτονικούς κόμβους, εξετάζει τους γειτονικούς τους κόμβους που δεν έχουν εξερευνηθεί κ.ο.κ. μέχρι να φθάσει σε κάποιο στόχο.

Διερευνώνται δηλαδή όλες οι διαδρομές μήκους N, πριν διερευνηθεί μια διαδρομή μήκους N+1, αρχίζοντας από τις διαδρομές μήκους 1. Ο αλγόριθμος αυτός, οδηγεί σε βέλτιστη λύση, όπου βέλτιστη λύση σημαίνει «στη μικρότερη απόσταση από την αρχική κατάσταση». Η αναζήτηση σε πλάτος οδηγεί στο φαινόμενο της συνδυαστικής έκρηξης, δηλαδή οι απαιτήσεις της σε μνήμη και χρόνο διερεύνησης έχουν εκθετική πολυπλοκότητα. Η εφαρμογή αυτής της μεθόδου για την πλοήγηση ενός μεγάλου χώρου αναζήτησης δεν είναι ρεαλιστική.

Επίσης ο αλγόριθμος BFS είναι μια «τυφλή μέθοδος αναζήτησης» γιατί επεκτείνεται και εξετάζει όλους τους κόμβους ενός γράφου συστηματικά για εύρεση της λύσης χωρίς κάποιες κατευθυντήριες γραμμές που να τον καθοδηγούν, όπως και ο πιο πάνω αλγόριθμος. Με άλλα λόγια ερευνά εξαντλητικά ολόκληρο το γράφο χωρίς να λαμβάνει υπόψη του που μπορεί να βρίσκεται η λύση μέχρι να την βρει.



Σχήμα 3.3: Αναζήτηση σε πλάτος

3.4. Το κριτήριο του Ntafos

3.4.1. Επιπρόσθετοι ορισμοί

Πριν να ορίσω το κριτήριο Ntafos πρέπει να αναφέρω ακόμα ένα ορισμό. Το κριτήριο Ntafos και γενικά όλα τα κριτήρια που υπάρχουν πιο κάτω θεωρούν ότι γίνεται εφαρμογή ενός path selection criterion σ' ένα ***module,*** M. Δηλαδή υποθέτουμε ότι ένα ***module*** είναι ή ένα κυρίως πρόγραμμα (main program) ή ένα κομμάτι κάποιου υποπρογράμματος (single subprogram) το οποίο έχει μια μόνο είσοδο και μια μόνο έξοδο.

Ένα path selection criterion ή πιο απλά ένα κριτήριο, είναι μια δήλωση που αναθέτει μια τιμή αληθείας σε κάποιο ζευγάρι **(M,P)**, όπου M είναι το module και P είναι το υποσύνολο των PATHS(M). Ένα ζευγάρι (M,P) ικανοποιεί ένα κριτήριο C αν C(M,P)=true.

Επίσης όταν αναφέρομαι σε def2(x) ή d2(x) εννοώ ορισμό (definition) της μεταβλητής X στον κόμβο 2. Το ίδιο ισχύει και για το use2(x) ή u2(x) με την διαφορά τώρα ότι αναφέρομαι σε χρήση (use) της μεταβλητής X στον κόμβο 2.

3.4.2. Ορισμός κριτηρίου Ntafos

Ο Simeon Ntafos χρησιμοποιεί τους γράφους ροής δεδομένων για να επιλέξει τα μονοπάτια προς έλεγχο. Ορίζει μια τάξη κριτηρίων επιλογής μονοπατιών προς έλεγχο τα οποία ονομάζει Αναγκαίες k-Πλειάδες (Required k-Tuples) [19]. Τα κριτήρια ορίζουν ότι το σύνολο των μονοπατιών καλύπτει αλυσίδες εναλλασσόμενων ορισμών και χρήσεων τις οποίες ονομάζει k-dr interactions. Δηλαδή k-definition/ reference interaction όπως

έχει καθιερωθεί στην ορολογία. Μπορούμε να το βρούμε και ως df-chain δηλαδή data flow chain. Στην παρούσα εργασία θα αναφέρομαι σε k-dr interactions όπως είναι καθιερωμένο. Σε ένα k-dr interaction κάθε ορισμός φτάνει στην επόμενη χρήση στην αλυσίδα, η οποία χρήση γίνεται στον ίδιο κόμβο που γίνεται και ο επόμενος ορισμός. Έτσι το k-dr interaction διαδίδει πληροφορία μέσα στο υπο-μονοπάτι, το οποίο ονομάζεται **interaction subpath για το k-dr interaction**.

Το κριτήριο Required k-Tuples ορίζεται για $k \geq 2$. Για αυτές τις τιμές του k, ένα k-dr interaction είναι μια σειρά k=[$d_1(x_1)$,$u_2(x_1)$,…..,$d_{k-1}(x_{k-1})$,$u_k(x_{k-1})$] από k-1 ορισμούς και k-1 χρήσεις που παρουσιάζονται σε k διακριτούς κόμβους n_1,n_2,n_3,…,n_k. Σημειώνεται ότι παρόλο που οι κόμβοι πρέπει να είναι διακριτοί, οι μεταβλητές $x_1, x_2 \ldots x_{k-1}$ δεν χρειάζεται να είναι διακριτές. Ένα interaction υπό-μονοπάτι για k είναι ένα υπό-μονοπάτι p = (n_1) p_1….(n_{k-1}) p_{k-1} (n_k) τέτοιο ώστε για όλα τα i, 1≤ i <k ,το υπό-μονοπάτι p_i είναι def-clear μονοπάτι wrt x.

Το κριτήριο Required k-Tuples ικανοποιείται από το ζευγάρι (M,P) μόνο αν υπάρχει τουλάχιστον ένα interaction υπό-μονοπάτι στο P (P=υποσύνολο των PATHS(M)) για κάθε l-dr interaction στο G(M) όπου 2 ≤ l ≤ k. Επιπρόσθετα, το P πρέπει να εξετάσει καθορισμένα branches και loops με συγκεκριμένα είδη μονοπατιών. Για να εξεταστούν όλα τα branches των predicate use $u_l(x_{l-1})$ που τελειώνουν ένα l-dr interaction λ, το P πρέπει να περιέχει ένα υπό-μονοπάτι p (m) για κάθε διάδοχο m του κόμβου n_l, όπου το p είναι ένα interaction υπό-μονοπάτι για το λ. Για να εξεταστούν όλα τα loops, αν το L είναι το innermost loop που περιέχει το πρώτο definition ή την τελευταία αναφορά (last reference) ενός l-dr interaction λ, τότε το P πρέπει να περιέχει υπό-μονοπάτια που και τα δύο καλύπτουν το λ και εξετάζουν το L (loop) ένα ελάχιστο (minimal) και ένα μεγαλύτερο (larger) αριθμό φορών.

Σε αυτό το κριτήριο, οι ορισμοί και οι χρήσεις των μεταβλητών, συσχετίζονται σαν ένας αρχικός κόμβος (source) και ένα τελικός (sink) αντίστοιχα.

Ο τυπικός ορισμός του κριτηρίου που δόθηκε από τον Ntafos παρουσίαζε ένα πρόβλημα το οποίο εντοπίστηκε από τους Clarke, Podgurski, Richardson και Zeil. Το πρόβλημα ήταν ότι το κριτήριο Required k-Tuples δεν ύπαγε (subsume) το κριτήριο για μια πλειάδα λιγότερη, δηλαδή το Required (k-1)-Tuples. Οι Clarke, Podgurski, Richardson και Zeil, αφού εντόπισαν το λάθος, προχώρησαν σε ένα αυστηρότερο ορισμό ο οποίος εξασφάλιζε ότι το Required k-Tuples ύπαγε - υπερνικούσε (subsume) το Required (k-1)-Tuples.

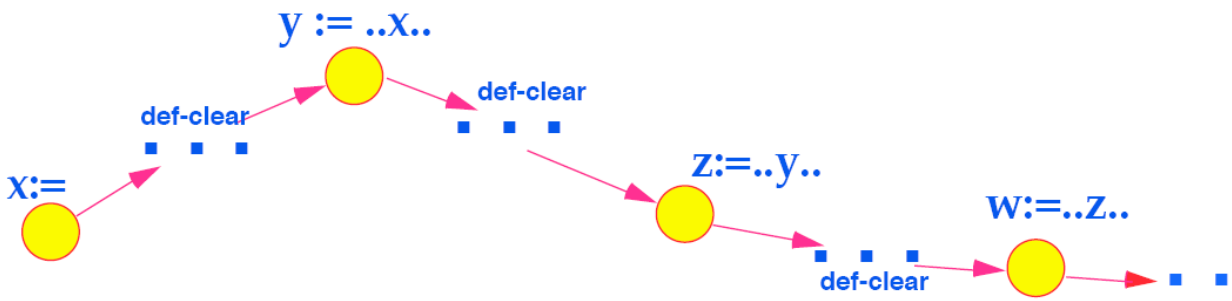
Το νέο κριτήριο Required k-Tuples που επαναπροσδιορίστηκε από τους Clarke, Podgurski, Richardson και Zeil είναι:

Έστω ότι το k είναι ένας σταθερός integer αριθμός, $k \geq 2$ (k είναι ο αριθμός των διακλαδώσεων).

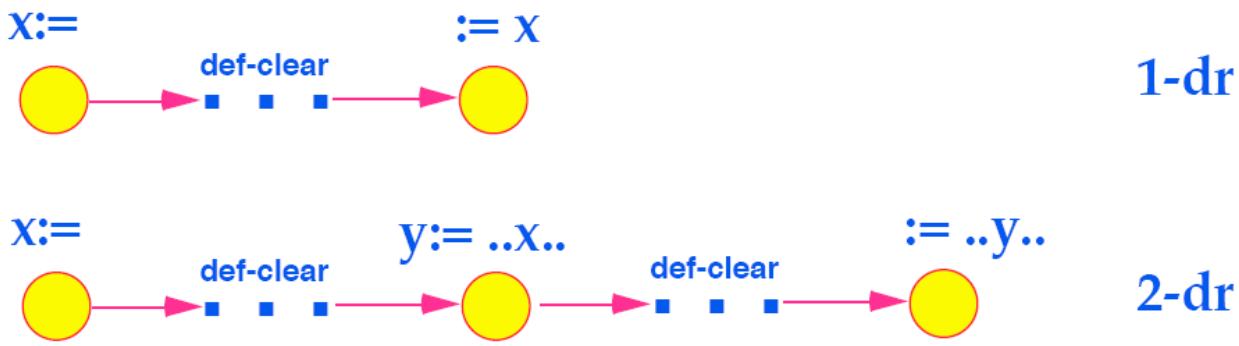
Το ζευγάρι (M,P) ικανοποιεί το κριτήριο Required k-Tuples αν για όλα τα l-dr interaction λ, στο G(M), $2 \leq l \leq k$, αν κάθε μια από τις ακόλουθες συνθήκες ισχύει:

1. Για όλους τους διάδοχους m του κόμβου n_l που συσχετίζονταν με την τελευταία χρήση (use) στο λ, το P πρέπει να περιέχει ένα υπο-μονοπάτι $p \cdot (m)$ τέτοιο ώστε το p να είναι ένα interaction subpath για το λ.
2. Αν ο κόμβος n_1 που συσχετιζόταν με την πρώτη δήλωση (first definition) στο λ βρίσκεται σε loop τότε το P πρέπει να περιέχει υπο-μονοπάτια $p = p_1 \cdot (n_1) \cdot p_2 \cdot p_3$ και $p' = p_1' \cdot (n_1) \cdot p_2' \cdot p_3'$ τέτοια ώστε: $(n_1) \cdot p_2 \cdot p_3$ και $(n_1) \cdot p_2' \cdot p_3'$ να ξεκινούν με interaction subpaths για το λ, το $p_1 \cdot (n_1) \cdot p_2$ να είναι cl-subpath για το loop L που περιέχει τον κόμβο n_1 και να διασχίζει το L ένα ελάχιστο (minimal) αριθμό φορών, και $p_1' \cdot (n_1) \cdot p_2'$ είναι cl-subpath για το L που διασχίζει το L ένα μεγαλύτερο (larger) αριθμό φορών.
3. Αν ο κόμβος n_l που συσχετίζεται με την τελευταία χρήση (use) στο λ βρίσκεται σε loop, τότε το P περιέχει υπο-μονοπάτια $p = p_1 \cdot p_2 \cdot (n_l) \cdot p_3$ και $p' = p_1' \cdot p_2' \cdot (n_l) \cdot p_3'$ τέτοια ώστε: $p_1 \cdot p_2 \cdot (n_l)$ και $p_1' \cdot p_2' \cdot (n_l)$ τελειώνουν με interaction subpaths για το λ, $p_2 \cdot (n_l) \cdot p_3$ είναι cl-subpath για το loop L που περιέχει τον κόμβο n_l και διασχίζει το L ένα ελάχιστο (minimal) αριθμό φορών, και $p_2' \cdot (n_l) \cdot p_3'$ είναι cl-subpath για το L που διασχίζει το L ένα μεγαλύτερο (larger) αριθμό φορών.

Το νέο κριτήριο Required k-Tuples που επαναπροσδιορίστηκε από τους Clarke, Podgurski, Richardson και Zeil διαφέρει ως προς εκείνο του Ntafos. Το κριτήριο που έδωσε ο Ntafos τα requires interaction υπο-μονοπάτια ήταν μόνο για το k-dr interaction ενώ το κριτήριο που επαναπροσδιορίστηκε πιο πάνω τα requires interaction υπο-μονοπάτια για κάθε l-dr interaction, περιείχαν και τα μονοπάτια όπου $l \leq k$. Έτσι αφού για κάθε module υπάρχει μια σταθερά n τέτοια ώστε δεν υπάρχουν k-dr interactions για $k > n$, στο κριτήριο του ο Ntafos δεν ύπαγε το required (k-1) tubles κριτήριο για ένα ακέραιο αριθμό $k > 2$. Προσέξτε ότι ο καινούριος ορισμός του κριτηρίου το διασφαλίζει αυτό.



Σχήμα 3.4: Γενικός ορισμός του κριτηρίου Ntafos



Σχήμα 3.5: Παράδειγμα του κριτηρίου Ntafos

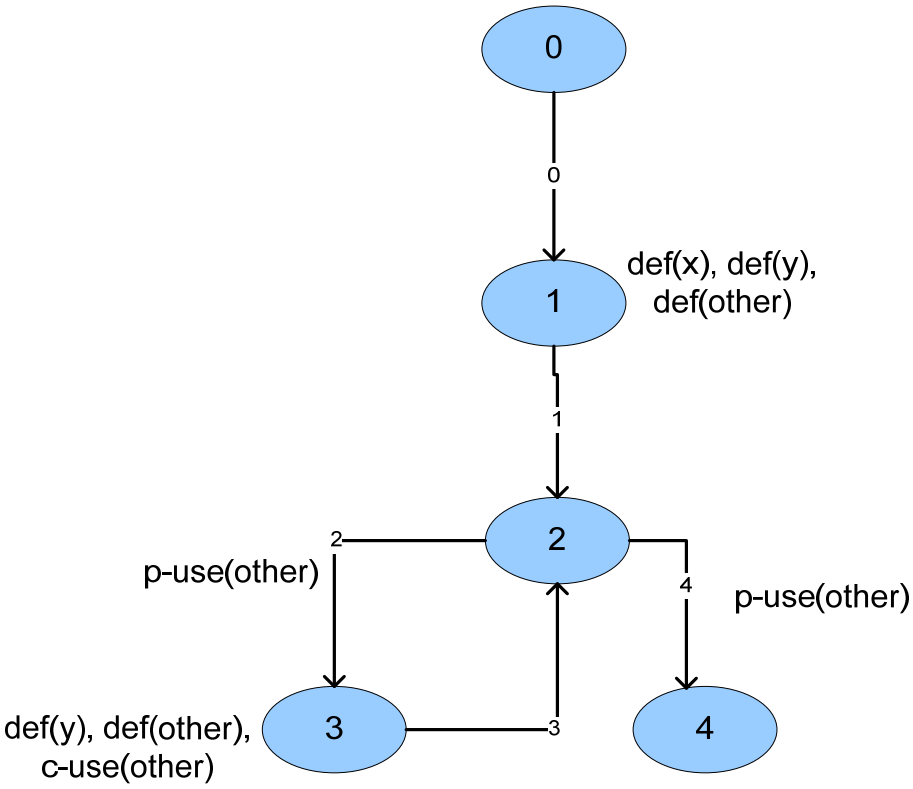
3.4.3. Πως ξέρω ποιο είναι το μέγιστο k-dr interaction

Πρακτικά είναι αδύνατο για μεγάλα προγράμματα να βρούμε ποιος αριθμός αντιπροσωπεύει το μέγιστο k αλυσίδας εναλλασσόμενων ορισμών και χρήσεων. Μεγάλα προγράμματα σημαίνει μεγάλοι και περίπλοκοι γράφοι ροής δεδομένων, οπότε όπως είναι φυσιολογικό θα είναι δύσκολη και επίπονη η εξερεύνηση τους για εύρεση του μέγιστου k.

Επίσης και για μικρότερα προγράμματα, η εύρεση του μέγιστου k είναι πάλι κάπως περίπλοκη αλλά κυρίως χρονοβόρα γιατί θα πρέπει να κοιτάζουμε για k=1,2,3,… μέχρι να βρούμε το μέγιστο. Η εύρεση του μέγιστου k μπορεί όμως να μην μας εξασφαλίζει απαραιτήτως επιτυχία αφού μπορεί μικρότερα k να μας αποκαλύπτουν λάθη στο πρόγραμμα μας. Οπόταν θα ήταν άδικη η προσπάθεια, ο χρόνος και ο κόπος εύρεσης του μέγιστου k διαπερνώντας τον γράφο πολλές φορές μέχρι να το βρούμε. Στην παρούσα εργασία το k επιλέγεται από το χρήστη.

3.4.4. Παράδειγμα κριτηρίου Ntafos

Για περισσότερη κατανόηση του κριτηρίου δίδω ένα παράδειγμα. Ας υποθέσουμε ότι έχουμε τον πιο κάτω γράφο του σχήματος 3.6 με τα definitions και uses που δεικνύονται στον γράφο.



Σχήμα 3.6: Γράφος ροής δεδομένων με τα αντίστοιχα defs και use

Από τον πιο πάνω γράφο θα εξηγήσω πως εξάγονται τα 1-dr interaction και 2-dr interaction. Με παρόμοιο τρόπο εξάγονται και μεγαλύτερου βαθμού dr interactions. Πριν συνεχίσω να πω ότι στον κόμβο 2 έχουμε predicate use τα οποία αντιπροσωπεύονται από τις ακμές που εξέρχονται του κόμβου 2 γι' αυτό το λόγο τα interactions που καταλήγουν στο συγκεκριμένο use πρέπει να καλύψουν και τα δύο μονοπάτια. Στην συγκριμένη περίπτωση και το μονοπάτι που θα περιέχει το ….2, 3 και το μονοπάτι που θα περιέχει το ….2, 4.

Για την αλυσίδα 1-dr interaction παίρνω ένα definition μιας μεταβλητής κάθε φορά και προχωρώντας στον γράφο κοιτάζω να βρω χρήση της συγκεκριμένης μεταβλητής σε κάποιο κόμβο. Το μονοπάτι που διασχίζω όμως από το definition της μεταβλητής μέχρι την χρήση της πρέπει να είναι def-clear (επεξηγήθηκε στην ενότητα 3.2.3). Κατά αυτό τον τρόπο καταλήγω στα ακόλουθα 1-dr interactions:

a. d1(other) στο p-u3(other)
b. d1(other) στο p-u4(other)
c. d1(other) στο c-u3(other)
d. d3(other) στο p-u2(other)
e. d3(other) στο c-u3(other)
f. d3(other) στο p-u4(other)

Τα μονοπάτια που παράχθηκαν από τα πιο πάνω interactions είναι τα ακόλουθα:

- 1, 2, 3 : ικανοποιεί το interaction a
- 1, 2, 4 : ικανοποιεί το interaction b
- 1, 2, 3 : ικανοποιεί τα interactions c
- 3, 2, 3 : ικανοποιεί το interaction d
- 3, 2, 3 : ικανοποιεί το interaction e
- 3, 2, 4 : ικανοποιεί το interaction f

Για την δημιουργία της αλυσίδας 2-dr interaction τώρα, λειτουργώ με τον ίδιο τρόπο, παίρνω δηλαδή ένα definition μιας μεταβλητής κάθε φορά και προχωρώντας στον γράφο κοιτάζω να βρω χρήση της συγκεκριμένης μεταβλητής σε κάποιο κόμβο (μέχρι εδώ λογαριάζεται ως 1-dr). Σ' αυτό το σημείο, στον ίδιο κόμβο που βρήκα την χρήση της συγκεκριμένης μεταβλητής, παίρνω ένα ορισμό όχι κατά ανάγκη της ίδιας μεταβλητής και διασχίζοντας τον γράφο βρίσκω ένα επόμενο κόμβο στον οποίο γίνεται χρήση αυτής της μεταβλητής. Το μονοπάτι που διασχίζω όμως από το definition της μεταβλητής μέχρι την χρήση της πρέπει να είναι και πάλι def-clear. Κατά αυτό τον τρόπο καταλήγω στα ακόλουθα 2-dr interactions για τον πιο πάνω γράφο:

a. d1(other), c-u3(other), d3(other), p-u4(other)
b. d1(other), c-u3(other), d3(other), p-u2(other)
c. d3(other), c-u3(other), d3(other), p-u2(other)
d. d3(other), c-u3(other), d3(other), p-u4(other)
e. d1(other), c-u3(other), d3(other), c-u3(other)
f. d3(other), c-u3(other), d3(other), c-u3(other)

Τα μονοπάτια που παράχθηκαν από τα πιο πάνω interactions είναι τα ακόλουθα:

- 1, 2, 3, 2, 4 : ικανοποιεί το interaction a
- 1, 2, 3, 2, 3 : ικανοποιεί τα interaction b

- 3, 2, 3, 2, 3 : ικανοποιεί το interaction c
- 3, 2, 3, 2, 4 : ικανοποιεί το interaction d
- 1, 2, 3, 2, 3 : ικανοποιεί το interaction e
- 3, 2, 3, 2, 3 : ικανοποιεί το interaction f

Να επισημάνω ότι τώρα που η αλυσίδα είναι 2-dr interaction, το loop πρέπει να διαπεραστεί ένα minimal αριθμό φορών 1 και ένα maximal αριθμό φορών 2. Φυσικά αν μπορεί να διαπεραστεί και εφόσον οι αλυσίδες που ακολουθεί είναι def-clear. Υπάρχει και η περίπτωση να έχουμε loop σε ένα πρόγραμμα και 2-dr interaction ή μεγαλύτερου βαθμού και το κριτήριο του Ntafos να εντοπίσει μόνο 1-dr interaction μονοπάτια λόγω του ότι οι αλυσίδες που ακολουθούνται δεν είναι def-clear.

Επιπρόσθετα με τα πιο πάνω μονοπάτια θα έχω και τα μονοπάτια που παράχθηκαν για 1-dr interaction αφού κάθε k-dr interaction υπάγει και τα μικρότερου βαθμού interaction ((k-1)-dr interaction κ.ο.κ.). Κατά τον ίδιο τρόπο παράγω και μεγαλύτερου βαθμού k-dr interactions αν υπάρχουν.

3.5. Τα κριτήρια των Rapps και Weyuker

Οι Rapps και Weyuker όρισαν μια οικογένεια από κριτήρια επιλογής μονοπατιών (path selection criteria) και ανάλυσαν τα κριτήρια αυτά σε μια προσπάθεια τους να καθορίσουν τις σχέσεις που υπάρχουν ανάμεσα στα μέλη της οικογένειας [19]. Αυτή η οικογένεια κριτηρίων περιλαμβάνει τρία γνωστά κριτήρια ελέγχου ροής (control flow criteria) και μερικά νέα κριτήρια επιλογής μονοπατιών (path selection criteria) βασισμένα στην ιδέα του data flow analysis. Για την παρουσίαση της οικογένειας κριτηρίων δεν χρειάζονται άλλοι ορισμοί εκτός από αυτούς που δόθηκαν στην παράγραφο 3.2 (βασικές έννοιες).

Τα κριτήρια ελέγχου ροής είναι το **all-paths** (path coverage), το **all-edges** (branch coverage) και το **all-nodes** (statement coverage). Για το κριτήριο all-paths πρέπει το σύνολο paths να περιέχει κάθε path μέσα στο control flow graph του module. Για τα κριτήρια all-edges και all-nodes πρέπει το σύνολο από paths να καλύπτει κάθε ακμή και κάθε κόμβο αντίστοιχα.

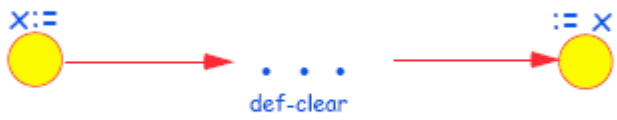
1. **All-Paths**: Το P ικανοποιεί το κριτήριο All-Paths αν το P περιλαμβάνει κάθε πλήρες μονοπάτι του G(M). Σημειώνεται ότι λόγω των βρόγχων επανάληψης, μερικά προγράμματα έχουν άπειρα μονοπάτια.
2. **All Edges**: Το P ικανοποιεί το κριτήριο All Edges αν κάθε ακμή του G(M) περιλαμβάνεται στο P.

3. **All Nodes:** Το P ικανοποιεί το κριτήριο All Nodes αν κάθε κόμβος του G(M) περιλαμβάνεται στο P.

Το all-paths υπάγει το all-edges το οποίο υπάγει all-nodes. Για τα περισσότερα modules, τα ζευγάρια (M,P) που ικανοποιούν το κριτήριο all-paths είναι αυτά που το P (σύνολο τους από μονοπάτια) είναι άπειρο. Σε τέτοιες περιπτώσεις το module σίγουρα περιέχει loop. Έτσι το κριτήριο all-paths δεν είναι χρήσιμο για τέτοια modules που περιέχουν loops. Τα κριτήρια όμως all-edges και all-nodes δεν απαιτούν σημαντικούς συνδυασμούς από κόμβους και ακμές. Τα υπόλοιπα κριτήρια Rapps και Weyuker διαχωρίζουν αυτούς τους συνδυασμούς που θεωρούνται σημαντικοί για τη ροή των δεδομένων διαμέσου ενός module. Αυτά τα κριτήρια επιλέγουν definition-clear subpaths μεταξύ του ορισμού (definition) και της χρήσης (use) που φθάνουν από εκείνο τον ορισμό.

Οι Rapps και Weyuker πρώτα όρισαν ένα κριτήριο που αναγκάζει κάθε ορισμό να χρησιμοποιηθεί. Το κριτήριο all-defs απαιτεί ότι το σύνολο paths πρέπει να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε κάποια χρήση που φθάνει από αυτό τον ορισμό.

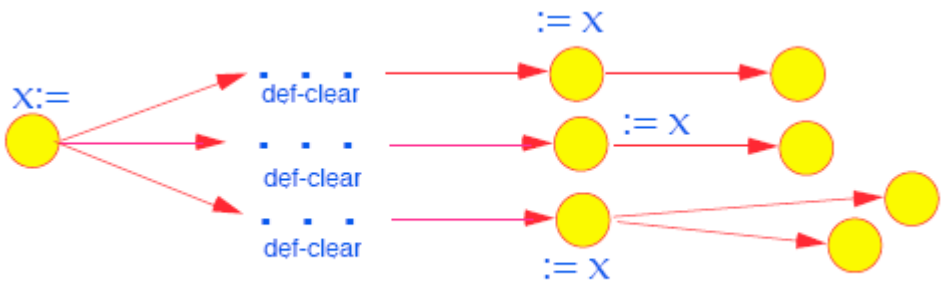
4. **All Defs**: Το P ικανοποιεί το κριτήριο All Definitions (all defs) αν για κάθε κόμβο i του G(M) και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **κάποιο** στοιχείο του dcu(i, x) ή dpu(i,x).



Σχήμα 3.7: Παράδειγμα του κριτηρίου all-defs.

Μετά οι Rapps και Weyuker όρισαν ένα κριτήριο που απαιτεί ότι όλες οι χρήσεις μεταβλητών που φθάνουν από ένα ορισμό πρέπει να καλυφθούν. Το κριτήριο all-uses απαιτεί το σύνολο μονοπατιών του να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε κάθε χρήση που φθάνει από αυτό τον ορισμό και κάθε διάδοχο κόμβο που περιέχει χρήση της συγκεκριμένης μεταβλητής.

5. **All Uses**: Το P ικανοποιεί το κριτήριο All Uses αν για κάθε κόμβο i του G και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dcu(i, x) και σε **όλα** τα στοιχεία του dpu(i,x).



Σχήμα 3.8: Παράδειγμα του κριτηρίου all-uses.

Ακολούθως οι Rapps και Weyuker όρισαν τρία κριτήρια που είναι παρόμοια με το all-uses αλλά διαχωρίζουν τα computational uses και τα predicate uses. Το πρώτο είναι το All C-Uses/Some P-Uses που απαιτεί το σύνολο μονοπατιών να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε κάθε computational use που φθάνει από αυτό τον ορισμό. Αν ο ορισμός φθάνει μόνο σε predicate uses, τότε το μονοπάτι πρέπει να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από τον ορισμό σε κάποιο predicate use.

6. **All C-Uses/Some P-Uses**: Το P ικανοποιεί το κριτήριο All C-Uses / Some P-Uses αν για κάθε κόμβο i του G(M) και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dcu(i,x). Εάν το dcu(i, x) είναι κενό, τότε το P πρέπει να περιλάβει ένα def-clear μονοπάτι ως προς το x από το i σε κάποια ακμή που περιλαμβάνεται στο dpu(x, i). Αυτό το κριτήριο απαιτεί ότι κάθε c-use μιας μεταβλητής x που ορίζεται στον κόμβο i, πρέπει να συμπεριληφθεί σε κάποιο μονοπάτι του P. Εάν δεν υπάρχει τέτοιο c-use, τότε πρέπει να συμπεριληφθεί κάποιο p-use του ορισμού της x. Κατά συνέπεια για να ικανοποιηθεί αυτό το κριτήριο, κάθε ορισμός που θα γίνει κάποτε, πρέπει να συμπεριλαμβάνει μια χρήση στα μονοπάτια του P, με έμφαση στα c-uses.

Κατά παρόμοιο τρόπο όρισαν και το κριτήριο All P-Uses/Some C-Uses. Με την διαφορά ότι το σύνολο μονοπατιών του απαιτεί να περιέχει τουλάχιστον ένα definition-

clear υπό-μονοπάτι από κάθε ορισμό σε κάθε predicate use που φθάνει από αυτό τον ορισμό και κάθε διάδοχο αυτής της χρήσης. Αν ο ορισμός φθάνει μόνο σε computation uses, τότε το μονοπάτι πρέπει να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από τον ορισμό σε κάποιο computation use.

7. **All P-Uses/Some C-Uses**: Το P ικανοποιεί το κριτήριο All P-Uses / Some C-Uses αν για κάθε κόμβο i του G(M) και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dpu(i,x). Εάν το dpu(i, x) είναι κενό, τότε το P πρέπει να περιλάβει ένα def-clear μονοπάτι ως προς το x από το i σε κάποιο κόμβο στο dcu(x, i). Όπως και στο προηγούμενο κριτήριο για να ικανοποιηθεί αυτό το κριτήριο, κάθε ορισμός ο οποίος θα χρησιμοποιηθεί κάποτε, πρέπει να χρησιμοποιηθεί σε ένα από τα μονοπάτια του P, αλλά αυτή τη φορά με έμφαση στα p-uses.

Το κριτήριο All P-Uses τώρα απαιτεί το σύνολο μονοπατιών του να περιέχει definition-clear υπό-μονοπάτια από κάθε ορισμό σε κάθε predicate use που φθάνει απ' αυτόν τον ορισμό και κάθε διάδοχο που κάνει use την συγκεκριμένη μεταβλητή.

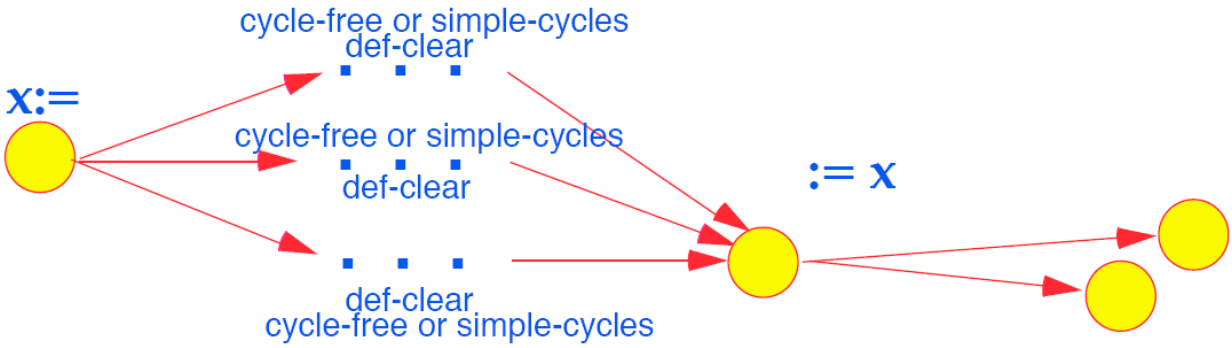
8. **All P-Uses**: Το P ικανοποιεί το κριτήριο All P-Uses αν για κάθε κόμβο i του G(M) και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dpu(i, x).

Το κριτήριο All C-Uses τώρα απαιτεί το σύνολο μονοπατιών του να περιέχει definition-clear υπό-μονοπάτια από κάθε ορισμό σε κάθε computation use που φθάνει απ' αυτόν τον ορισμό και κάθε διάδοχο που κάνει use την συγκεκριμένη μεταβλητή.

9. **All C-Uses**: Το P ικανοποιεί το κριτήριο All C-Uses αν για κάθε κόμβο i του G και κάθε $x \in def(i)$, το P περιλαμβάνει ένα def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dcu(i, x).

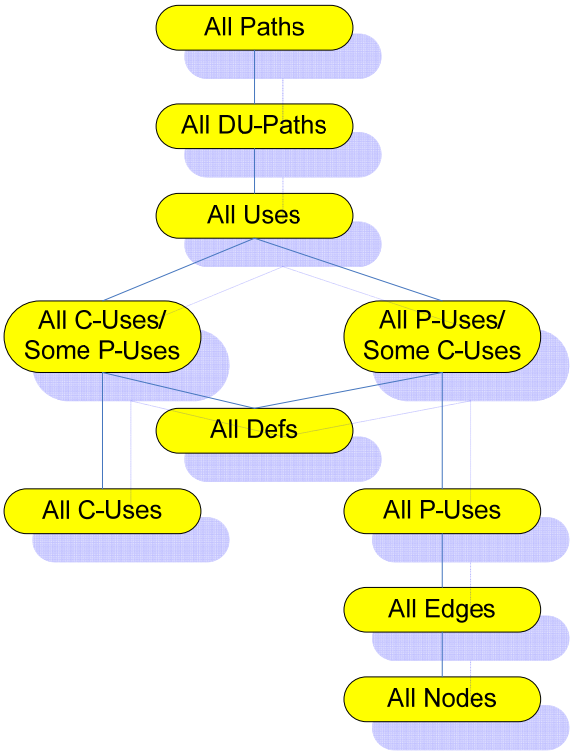
Τελευταίο κριτήριο είναι το all-DU-Paths. Το κριτήριο αυτό πάει ένα βήμα πέρα από το κριτήριο all-uses. Εκτός του ότι απαιτεί ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε όλους τους διάδοχους για κάθε χρήση που φτάνει απ' αυτόν τον ορισμό, το κριτήριο all-DU-Paths απαιτεί κάθε definition-clear υπό-μονοπάτι να είναι simple-cycle ή cycle-free. Αυτός ο περιορισμός προστέθηκε για να επιβεβαιώσει ότι το σύνολο μονοπατιών είναι πεπερασμένο.

10. **All DU-Paths**: Ορίζουμε ένα μονοπάτι $(n_1, n_2 \ldots n_n)$ σαν άκυκλο (loop-free) αν και μόνο αν $n_i \neq n_j$ για κάθε $i \neq j$. Το P ικανοποιεί το κριτήριο All DU-Paths αν για κάθε κόμβο i του G(M) και κάθε $x \in def(i)$, το P περιλαμβάνει κάθε loop-free, def-clear μονοπάτι ως προς το x από το i σε **όλα** τα στοιχεία του dcu(i, x) και σε **όλα** τα στοιχεία του dpu(i, x). Σημειώνεται ότι το σύνολο των μονοπατιών του γράφου δεν χρειάζεται να είναι loop-free.



Σχήμα 3.9: Παράδειγμα του κριτηρίου All DU-Paths

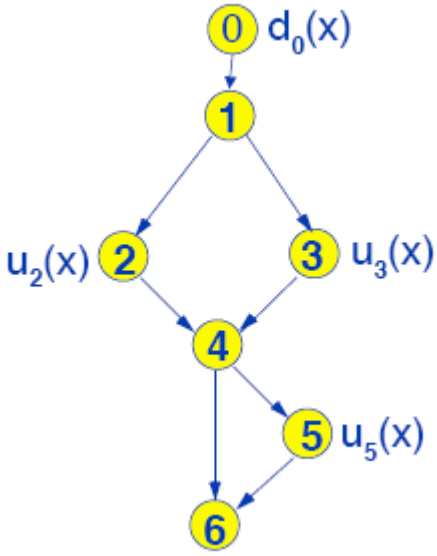
Στο [16] παρέχεται μια πλήρης ανάλυση ως προς το ποιο κριτήριο είναι πιο αυστηρό από το άλλο. Τα κριτήρια κατατάσσονται σε μια σειρά ανάλογα με την αυστηρότητά τους. Μπορεί κάποιος να πει ότι όταν ικανοποιείται ένα αυστηρότερο κριτήριο τότε αυτόματα ικανοποιούνται και τα ασθενέστερα. Η απόδειξη για το πόσο ισχυρά και αυστηρά είναι τα κριτήρια που παρουσιάστηκαν καθώς και η διάταξη τους με βάση τα κριτήρια αυτά παρουσιάζονται στο [16]. Το ακόλουθο σχήμα παρουσιάζει την διάταξη των κριτηρίων.



Σχήμα 3.10: Διάταξη των κριτηρίων Rapps και Weyuker

3.5.1. Μερικά παραδείγματα κάποιων κριτηρίων

Για περισσότερη κατανόηση των κριτηρίων Rapps και Weyuker δίδω κάποια παραδείγματα. Ας υποθέσουμε ότι έχουμε τον παρακάτω γράφο του σχήματος 3.11 με τα definitions και uses που δεικνύονται στον γράφο.



Σχήμα 3.11: Γράφος δεδομένων με τα αντίστοιχα defs και use

3.5.1.1. Παράδειγμα All Definitions (All-Defs)

Το κριτήριο All Definitions απαιτεί ότι το σύνολο paths πρέπει να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε κάποια χρήση που φθάνει από αυτό τον ορισμό.

Άρα με βάση το πιο πάνω γράφο έχουμε ένα ορισμό στον κόμβο μηδέν, οπόταν το κριτήριο **απαιτεί ένα μονοπάτι από το d0(x) σε κάποιο use**. Ένα ικανοποιητικό μονοπάτι είναι το 0, 1, 2, 4, 6.

3.5.1.2. Παράδειγμα all uses

Το κριτήριο all-uses απαιτεί το σύνολο μονοπατιών του να περιέχει τουλάχιστον ένα definition-clear υπό-μονοπάτι από κάθε ορισμό σε κάθε χρήση που φθάνει από αυτό τον ορισμό και κάθε διάδοχο κόμβο που περιέχει χρήση της συγκεκριμένης μεταβλητής.

Για το συγκεκριμένο παράδειγμα έχουμε ότι το κριτήριο απαιτεί:

- Από το d0(x) προς το u2(x)
- Από το d0(x) προς το u3(x)
- Από το d0(x) προς το u5(x)

Άρα τα ικανοποιητικά μονοπάτια για το κριτήριο αυτό είναι:

- 0, 1, 2, 4, 5, 6
- 0, 1, 3, 4, 6

Το πρώτο μονοπάτι μας καλύπτει για τα use2(x) και use5(x) και το δεύτερο για το use3(x).

3.5.1.3. Παράδειγμα all-DU-paths

Το κριτήριο all-DU-Paths απαιτεί ένα definition-clear υπό-μονοπάτι από κάθε ορισμό προς όλους τους διάδοχους που περιέχουν χρήση και φτάνουν απ' αυτόν τον ορισμό, επίσης απαιτεί κάθε definition-clear υπό-μονοπάτι να είναι simple-cycle ή cycle-free.

Για το συγκεκριμένο παράδειγμα έχουμε ότι το κριτήριο απαιτεί:

- Από το d0(x) προς το u2(x)
- Από το d0(x) προς το u3(x)
- Και τα δύο μονοπάτια από το d0(x) προς το u5(x)

Άρα τα ικανοποιητικά μονοπάτια για το κριτήριο αυτό είναι:

- 0, 1, 2, 4, 5, 6
- 0, 1, 3, 4, 5, 6

Το πρώτο μονοπάτι μας καλύπτει τα use2(x) και use5(x) και το δεύτερο τα use3(x) και use5(x).

3.6. Τα κριτήρια Laski και Korel

Οι Laski και Korel στο [20] προχώρησαν στην πρόταση κριτηρίων με βάση τους γράφους ροής δεδομένων. Τα κριτήρια που έχουν προτείνει βασίζονται στις αλυσίδες ορισμού – χρήσης (def – use chains) μια μεταβλητής, με σκοπό της καθοδήγηση του ελέγχου ενός προγράμματος. Η θεμελίωση των κριτηρίων στηρίζεται σε συνδυασμούς ορισμών οι οποίοι φτάνουν στο σημείο χρήσης της μεταβλητής μέσω κάποιου μονοπατιού.

3.6.1. Επιπρόσθετοι ορισμοί

Στην προτεινόμενη στρατηγική τους, καθορίζονται δύο διαφορετικά κριτήρια εκ τον οποίων το ένα μπορεί να χωριστεί περαιτέρω σε δύο υποκατηγορίες. Προτού παρουσιάσουμε τα κριτήρια θα δώσουμε κάποιους ορισμούς που απαιτούνται για την πλήρη κατανόησή τους. Και τα δύο κριτήρια είναι βασισμένα στην παρατήρηση ότι, σε οποιοδήποτε δεδομένο κόμβο, μπορούν να υπάρξουν χρήσεις διαφορετικών μεταβλητών, όπου ο ορισμός (def) κάθε μεταβλητής έγινε σε προηγούμενους, διαφορετικούς μεταξύ τους, κόμβους.

Ένας ορισμός στον κόμβο i θεωρείται ζωντανός σε έναν κόμβο j εάν δεν υπάρχει κανένας επαναπροσδιορισμός (νέο def) της μεταβλητής μεταξύ των κόμβων i και j. Το σύνολο των ζωντανών ορισμών όλων των μεταβλητών εισόδου σε ένα κόμβο i, ονομάζεται «περιβάλλον δεδομένων» (**data environment**), και συμβολίζεται ως DE(i). Η εκτέλεση μιας εντολής με n ορίσματα απαιτεί την ταυτόχρονη χρήση μιας πλειάδας από n ορίσματα μεταβλητών εισόδου από το data environment. Το γεγονός αυτό εκφράζεται από τον ορισμό των **data context**. Με τον ορισμό **elementary data context** μιας εντολής i, εννοούμε ένα σύνολο ορισμών των μεταβλητών που μετέχουν στην εντολή, τέτοιο ώστε (α) να υπάρχει ένα μονοπάτι ελέγχου από την αρχή του προγράμματος μέχρι την εντολή i και (β) όλοι οι ορισμοί του συνόλου να είναι ζωντανοί όταν το μονοπάτι φτάσει στο i. Το data context DC(i) μιας εντολής i είναι το σύνολο όλων των elementary

data context της εντολής. Με τον ορισμό **ordered data context** λογαριάζουμε και την σειρά που τα definitions λαμβάνουν χώρα.

Πιο κάτω δίνω μερικά παραδείγματα των ορισμών που επεξήγησα πιο πάνω για περισσότερη κατανόηση του αναγνώστη.

```
1  read(x);        d1(x)
2  y = 1;          d2(y)
3  z = y;          d3(z)
4  while (x > 0)
   {
5    if (x - y > 0)
6      y = y + 1;  d6(y)
     else
7      x = 1;      d7(x)
8    z = y / x;    d8(z)
   }
9  write(z);
```

$DE(3) = \{ d_2(y) \},$
$DE(4) = \{ d_1(x), d_7(x) \},$
$DE(5) = \{ d_1(x), d_2(y), d_7(x), d_6(y) \},$
$DE(6) = \{ d_2(y), d_6(y) \},$
$DE(8) = \{ d_1(x), d_2(y), d_7(x), d_6(y) \},$
$DE(9) = \{ d_3(z), d_8(z) \}$

Σχήμα 3.12: Παράδειγμα ορισμού data environment

```
1  read(x);        d1(x)
2  y = 1;          d2(y)
3  z = y;          d3(z)
4  while (x > 0)
   {
5    if (x-y > 0)
6      y = y + 1; d6(y)
     else
7      x = 1;     d7(x)
8    z = y/x;     d8(z)
   }
9  write(z);
```

$DC(3) = \{ (d_2(y)) \}$

$DC(4) = \{ (d_1(x)), (d_7(x)) \}$

$DC(5) = \{ (d_1(x), d_2(y)), (d_1(x), d_6(y)),$
$(d_7(x), d_2(y)), (d_7(x), d_6(y)) \}$

$DC(6) = \{ (d_2(y)), (d_6(y)) \}$

$DC(8) = \{ (d_1(x), d_6(y)), (d_7(x), d_2(y)),$
$(d_7(x), d_6(y)) \}$

$DC(9) = \{ (d_3(z)), (d_8(z)) \}$

Σχήμα 3.13: Παράδειγμα ορισμού data context

```
1  read(x);        d1(x)
2  y = 1;          d2(y)
3  z = y;          d3(z)
4  while (x > 0)
   {
5    if (x - y > 0)
6      y = y + 1;  d6(y)
     else
7      x = 1;      d7(x)
8    z = y / x;    d8(z)
   }
9  write(z);
```

For example:

$ec(8) = (d_1(x), d_6(y))$

Σχήμα 3.14: Παράδειγμα ορισμού elementary data context

```
1  read(x);        d1(x)
2  y = 1;          d2(y)
3  z = y;          d3(z)
4  while (x > 0)
   {
5    if (x-y > 0)
6      y = y + 1;  d6(y)
     else
7      x = 1;      d7(x)
8    z = y/x;      d8(z)
   }
9  write(z);
```

For example:

$DC(8) = \{ (d_1(x), d_6(y)),$
$(d_7(x), d_2(y)),$
$(d_7(x), d_6(y)) \}$

$ODC(8) = \{ [d_1(x), d_6(y)],$
$[d_2(y), d_7(x)],$
$[d_7(x), d_6(y)],$
$[d_6(y), d_7(x)] \}$

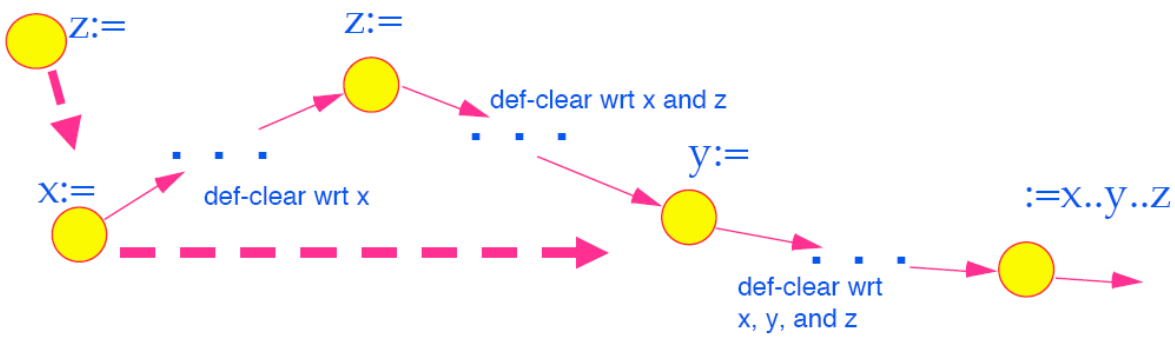
Σχήμα 3.15: Παράδειγμα ορισμού ordered data context

3.6.2. Τα κριτήρια των Laski και Korel

Οι Laski και Korel όρισαν τρία διαφορετικά κριτήρια επιλογής μονοπατιών βασισμένα στο data flow analysis. Θα αναφερόμαστε σ' αυτά ως το Reach Coverage criterion (Στρατηγική 1), το Context Coverage criterion (Στρατηγική 2) και το Ordered Context Coverage criterion (αλλαγμένη η στρατηγική 2).

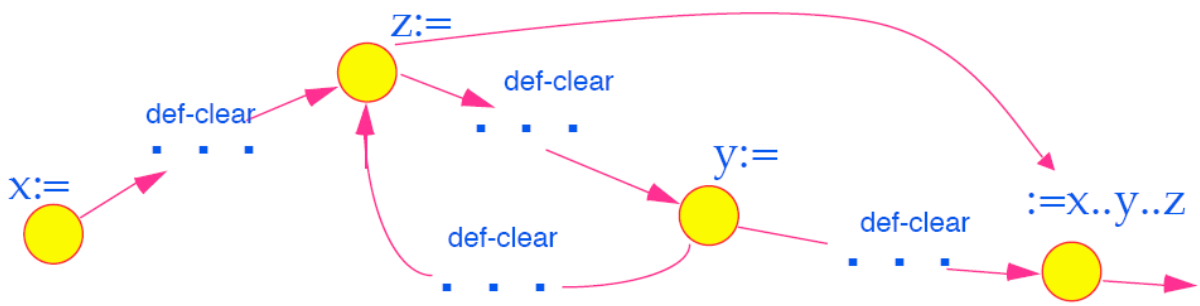
Το πρώτο κριτήριο, το Reach Coverage ικανοποιεί το (M, P) αν για όλα τα definitions $d_m(x)$ και όλα τα uses $u_n(x)$ που φθάνουν από αυτά τα definitions, το P περιέχει τουλάχιστον ένα μονοπάτι (m) p (n) τέτοιο ώστε το p να είναι definition clear wrt x. Με άλλα λόγια το πρώτο κριτήριό τους, απαιτεί όπως κάθε χρήση της μεταβλητής στους κόμβους στους οποίους ο ορισμός της μεταβλητής είναι ζωντανός, να ελεγχθεί τουλάχιστον μία φορά. Βασικά αναζητείται η κάλυψη των μονοπατιών από κάθε ορισμό σε κάθε χρήση στην οποία φτάνει εκείνος ο ορισμός χωρίς να ξανά ορίζεται.

Το δεύτερο κριτήριο, το Context Coverage criterion ικανοποιεί το (M, P) αν για όλα τα definitions context DC(n), το P περιέχει τουλάχιστον ένα context υπό-μονοπάτι για το DC(n). Δηλαδή κάθε elementary data context της κάθε εντολής να ελεγχθεί τουλάχιστο μια φορά. Εδώ αναζητούνται υπό-μονοπάτια στα οποία κάθε **σύνολο** από ορισμούς φτάνει στην χρήση τους σε κάθε κόμβο μέσα από def-clear μονοπάτια. Αυτό παρουσιάζεται στο επόμενο σχήμα.



Σχήμα 3.16: Παράδειγμα του 2ου κριτηρίου Laski – Korel (Context Coverage)

Το τρίτο και τελευταίο κριτήριο, το Ordered Context Coverage criterion ικανοποιεί το (M, P) αν για όλα τα ordered definitions context ODC(n), το P περιέχει τουλάχιστον ένα ordered context υπομονοπάτι για το ODC(n). Ουσιαστικά είναι μια παραλλαγή και μια πιο αυστηρή διατύπωση του πιο πάνω κριτηρίου που απαιτεί την ύπαρξη υπό-μονοπατιού στο οποίο κάθε σειρά από ορισμούς φτάνει σε κάθε κόμβο όπου γίνεται χρήση τους. Το σχήμα που ακολουθεί παρουσιάζει αυτή την απαίτηση.



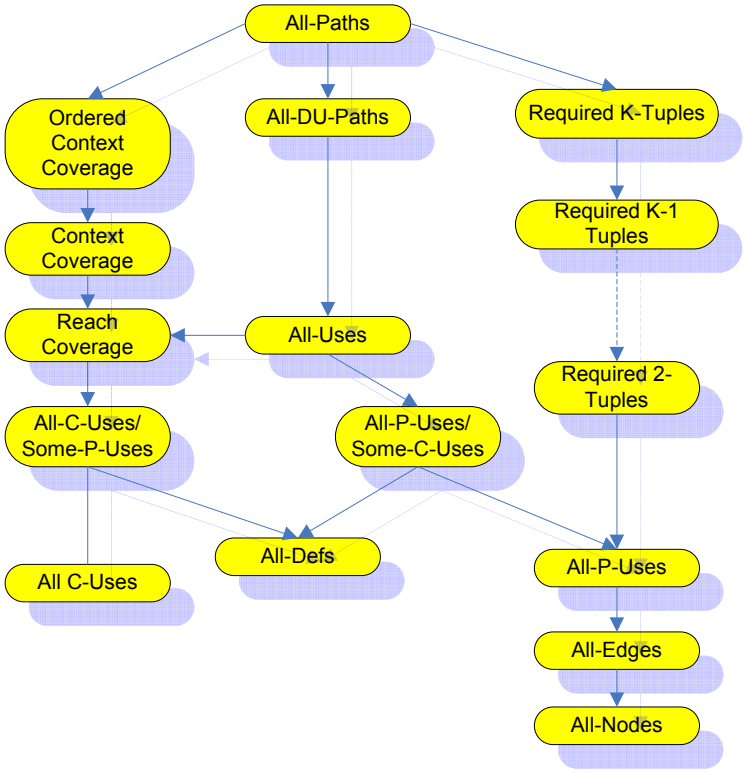
Σχήμα 3.17: Παράδειγμα του Ordered Context Coverage

3.7. Σύγκριση των κριτηρίων

Τα κριτήρια που ανάφερα πιο πάνω έγιναν αντικείμενο σχολιασμού από κάποιους ερευνητές. Για να οριστεί τι είναι ένα πετυχημένο κριτήριο έγιναν μερικές αποτιμήσεις. Κάποιες από τις συγκρίσεις ασχολήθηκαν με το τι τελικά πετυχαίνει το κάθε κριτήριο ενώ άλλες προχώρησαν σε συνδυασμούς κριτηρίων με σκοπό να παραχθεί η ιεράρχηση όλων των κριτηρίων.

Σε κάποια από τα άρθρα που δημοσιεύτηκαν οι συγγραφείς ξεκινούν με την παρουσίαση των κριτηρίων ενώ στη συνέχεια αναγνωρίζουν λάθη και παραλήψεις στα κριτήρια αυτά και παραθέτουν καινούρια αναθεωρημένα κριτήρια τα οποία υπερνικούν τα λάθη και παραλήψεις των προηγούμενων. Τέλος, προσπαθούν να δώσουν μια κοινή συνισταμένη σε όλα τα κριτήρια ώστε να μπορεί να γίνει σύγκριση μεταξύ τους. Η εργασία τους ολοκληρώνεται με την κατάταξη των κριτηρίων με βάση ποιο περικλείει (υπάγει) στο ορισμό του τα υπόλοιπα.

Ουσιαστικά επεκτείνουν την ιεράρχηση των κριτηρίων των Rapps και Weyuker (σχήμα 3.10) και ενσωματώνουν στην ιεραρχία τα κριτήρια από τις άλλες δύο προσπάθειες (σχήμα 3.18). Σύγκριση των κριτηρίων έγινε και από τον Ntafos στο [21] με περίπου τα ίδια αποτελέσματα, ενώ πολύ αργότερα στο [23] οι συγγραφείς έδωσαν παρουσίασαν και συγκρίνανε τα κριτήρια με βάση την γλώσσα περιγραφής λογισμικού Computational Tree Logic (CTL) η οποία παρέχει δυνατότητες περιγραφής της συμπεριφοράς στην εξέλιξη του χρόνου.



Σχήμα 3.18: Η τελική ιεράρχηση όλων των κριτηρίων

Κεφάλαιο 4

Γενετικοί αλγόριθμοι

4.1. Γενικά 69
4.2. Ιστορική αναδρομή γενετικών αλγορίθμων 70
4.3. Βιολογική ορολογία 71
4.4. Πως λειτουργούν.....73
4.5. Περιγραφή τρόπου υλοποίησης γενετικών αλγορίθμων.....73
4.6. Λόγοι που οδήγησαν στους γενετικούς αλγορίθμους 80
4.7. JGAP: Βιβλιοθήκη υλοποίησης γενετικών αλγορίθμων 83

4.1. Γενικά

Οι **Γενετικοί αλγόριθμοι** ανήκουν στο κλάδο της επιστήμης υπολογιστών και αποτελούν μια μέθοδο αναζήτησης βέλτιστων λύσεων σε συστήματα που μπορούν να περιγραφούν ως μαθηματικό πρόβλημα. Είναι χρήσιμοι σε προβλήματα που περιέχουν πολλές παραμέτρους/διαστάσεις και δεν υπάρχει αναλυτική μέθοδος που να μπορεί να βρει το βέλτιστο συνδυασμό τιμών για τις μεταβλητές ώστε το υπό εξέταση σύστημα να αντιδρά με όσο το δυνατόν με το θεμιτό τρόπο.

Ο τρόπος λειτουργίας των Γενετικών Αλγορίθμων είναι εμπνευσμένος από την βιολογία. Χρησιμοποιεί την ιδέα της εξέλιξης μέσω γενετικής μετάλλαξης, φυσικής επιλογής και διασταύρωσης. Οι Γενετικοί Αλγόριθμοι είναι αρκετά απλοί στην υλοποίησή τους. Οι τιμές για τις παραμέτρους του συστήματος πρέπει να κωδικοποιούνται με τρόπο ώστε να αναπαρασταθούν από μια μεταβλητή που περιέχει σειρά χαρακτήρων ή δυαδικών ψηφίων (0/1). Αυτή η μεταβλητή μιμείται το γενετικό κώδικα που υπάρχει στους ζωντανούς οργανισμούς. Αρχικά, ο Γενετικός Αλγόριθμος παράγει πολλαπλά αντίγραφα της μεταβλητής/γεννητικού κώδικα, συνήθως με τυχαίες τιμές, δημιουργώντας ένα πληθυσμό λύσεων. Κάθε λύση (τιμές για τις παραμέτρους του συστήματος) δοκιμάζεται για το πόσο κοντά φέρνει την αντίδραση του συστήματος στην επιθυμητή, μέσω μιας συνάρτησης που δίνει το μέτρο ικανότητας της λύσης και η οποία ονομάζεται συνάρτηση ικανότητας (fitness function).

Οι λύσεις που βρίσκονται πιο κοντά στην επιθυμητή, σε σχέση με τις άλλες, σύμφωνα με το μέτρο που μας δίνει η συνάρτηση ικανότητας, αναπαράγονται στην επόμενη γενιά λύσεων και λαμβάνουν μια τυχαία μετάλλαξη. Επαναλαμβάνοντας

αυτή τη διαδικασία για αρκετές γενιές, οι τυχαίες μεταλλάξεις σε συνδυασμό με την επιβίωση και αναπαραγωγή των γονιδίων/λύσεων που πλησιάζουν καλύτερα το επιθυμητό αποτέλεσμα θα παράγουν ένα γονίδιο/λύση που θα περιέχει τις τιμές για τις παραμέτρους που ικανοποιούν όσο καλύτερα γίνεται την συνάρτηση ικανότητας.

Υπάρχουν διάφορες εκδοχές της παραπάνω διαδικασίας για τους Γ.Α από τις οποίες κάποιες περιλαμβάνουν και τη διασταύρωση (ζευγάρωμα) γονιδίων/λύσεων ώστε ο αλγόριθμος να φτάσει στο αποτέλεσμα πιο γρήγορα. Καθώς υπάρχει το στοχαστικό (τυχαίο) συστατικό της μετάλλαξης και ζευγαρώματος, κάθε εκτέλεση του Γ.Α μπορεί να συγκλίνει σε διαφορετική λύση και σε διαφορετικό χρόνο. Η απόδοση του Γ.Α εξαρτάται επί το πλείστον από την συνάρτηση ικανότητας και συγκεκριμένα από το κατά πόσο το μέτρο της περιγράφει την βέλτιστη λύση. Οι γενετικοί αλγόριθμοι είναι ένα πεπερασμένο σύνολο οδηγιών για την εκπλήρωση ενός έργου, το οποίο δεδομένης μιας αρχικής κατάστασης θα οδηγήσει σε μια αναγνωρίσιμη τελική κατάσταση, και το οποίο προσπαθεί να μιμηθεί την διαδικασία της βιολογικής εξέλιξης. Οι γενετικοί αλγόριθμοι προσπαθούν να βρουν τη λύση ενός προβλήματος με το να προσομοιώνουν την εξέλιξη ενός πληθυσμού «λύσεων» του προβλήματος.

Είναι μια τεχνική προγραμματισμού που εισήγαγε στα τέλη της δεκαετίας του 1960 ο Τζον Χόλαντ, ερευνητής του Ινστιτούτου της Σάντα Φε (ΗΠΑ).

Οι γενετικοί αλγόριθμοι είναι μια από τις βάσεις των Προγραμμάτων Τεχνητής Ζωής. Συγκεκριμένα, επιχειρεί να αναπαράξει στους υπολογιστές τους μηχανισμούς της βιολογικής εξέλιξης με τον ίδιο τρόπο που η τεχνητή νοημοσύνη επιχειρεί να αναπαραστήσει και να μιμηθεί τις διαδικασίες της γνώσης.

Τα προγράμματα εξελίσσονται μέχρι να φτάσουν, μέσω μεταλλάξεων, διασταυρώσεων και φυσικής επιλογής, σε μια αποτελεσματική φόρμουλα η οποία θα εκτελεί με τον καλύτερο δυνατό τρόπο μια συγκεκριμένη εργασία.

4.2. Ιστορική αναδρομή γενετικών αλγορίθμων

Γύρω στο τέλος της δεκαετίας του 1950 και αρχές του 1960, άρχισαν να εμφανίζονται οι πρώτες περιπτώσεις αλγορίθμων που αργότερα και μέχρι σήμερα ονομάζονται γενετικοί (ΓΑ). Η αρχή έγινε όταν επιστήμονες πληροφορικής μελέτησαν ανεξάρτητα τα εξελικτικά συστήματα, με την ιδέα ότι η εξέλιξη θα μπορούσε να χρησιμοποιηθεί ως εργαλείο βελτιστοποίησης για τα προβλήματα εφαρμοσμένης μηχανικής. Η ιδέα σε όλα αυτά τα συστήματα ήταν να εξελιχθεί ο πληθυσμός των υποψηφίων λύσεων σε ένα δεδομένο πρόβλημα, χρησιμοποιώντας αρχές της εξελικτικής βιολογίας (την φυσική γενετική παραλλαγή και την φυσική επιλογή).

Οι ΓΑ αναπτύχθηκαν συστηματικά από τον John Holland και την ομάδα του στο πανεπιστήμιο του Michigan την δεκαετία του ’70. Ακολούθησαν πολλές εργασίες και από άλλους ερευνητές με αποτέλεσμα πολύ σύντομα να θεμελιωθεί και να πιστοποιηθεί η εγκυρότητα τους. Θεωρούνται αποτελεσματικοί σε προβλήματα βελτιστοποίησης συναρτήσεων, επιχειρησιακής έρευνας, μηχανικής, οικονομικών αλλά και σε εφαρμογές αυτόματου ελέγχου. Η γρήγορη και πλατιά διάδοση των Γ.Α. οφείλεται κυρίως στην απλότητα και την ευκολία υλοποίηση τους, στο ευρύ πεδίο εφαρμογών και στην σταθερότητα απόδοσης.

Ακολούθησαν μελέτες από όπου προέκυψαν διαφοροποιήσεις των Γ.Α. όπως: Η εξελισσόμενη Στρατηγική (Evolution strategy) από τους Ingo Rechenberg και Hans Paul Schawefel. Ο εξελισσόμενος προγραμματισμός (Evolutionary Programming) από τους Fogel, Owens και Walsh. Και ο γενετικά εξελισσόμενος Προγραμματισμός (Genetic programming) από τον John Koza.

Σήμερα όλες αυτές οι τεχνικές έχουν ενοποιηθεί κάτω από το όνομα Εξελισσόμενη **Υπολογιστική νοημοσύνη** (Evolutionary computation intelligence). Η υπολογιστική νοημοσύνη είναι ο διάδοχος της Τεχνητής νοημοσύνης. Η υπολογιστική νοημοσύνη συνδυάζει στοιχεία μάθησης (learning), προσαρμογής (adaptation), εξέλιξης (evolution) και ασαφούς λογικής (fuzzy logic) για να δημιουργήσει προγράμματα που είναι κατά κάποιο τρόπο «έξυπνα». Η παρούσα εργασία, της οποίας και ο τίτλος παρέχει τον ορισμό υπολογιστική νοημοσύνη, συνδυάζεται με τους όρους εξέλιξη και προσαρμογή οι οποίοι παρέχονται μέσω των γενετικών αλγορίθμων. Μερικές εφαρμογές της υπολογιστικής νοημοσύνης είναι οι εξής:

- Λήψη αποφάσεων σε περιβάλλον Αβεβαιότητας
- Κόστος κύκλου ζωής
- Τιμολόγηση αγαθών και υπηρεσιών
- Πρόβλεψη ζήτησης αγαθών και άλλων μεγεθών
- Χρονοπρογραμματισμός εργασιών και υπηρεσιών
- Αλγόριθμοι δρομολόγησης

4.3. Βιολογική ορολογία

Προτού προχωρήσω είναι χρήσιμο να εξηγήσω κάποια σχετική ορολογία που χρησιμοποιείται στους γενετικούς αλγόριθμους δανεισμένη από τη βιολογική ορολογία. Στα πλαίσια των γενετικών αλγορίθμων, αυτοί οι βιολογικοί όροι χρησιμοποιούνται στο πνεύμα της αναλογίας με την πραγματική βιολογία, αν και οι οντότητες στις οποίες αναφέρονται είναι πολύ απλούστερες από τις πραγματικές αντίστοιχες βιολογικές έννοιες.

Όλοι οι οργανισμοί διαβίωσης αποτελούνται από τα κύτταρα, και κάθε κύτταρο περιέχει το ίδιο σύνολο από ένα ή περισσότερα χρωμοσώματα (chromosomes - strings of DNA) τα οποία χρησιμεύουν ως ένα «σχεδιάγραμμα» (blueprint) για τον οργανισμό.

Ένα χρωμόσωμα μπορεί να είναι εννοιολογικά διαιρεμένο σε γονίδια(genes) όπου το κάθε ένα κωδικοποιεί μια ιδιαίτερη πρωτεΐνη(protein). Θα μπορούσαμε να σκεφτούμε ένα γονίδιο ως την κωδικοποίηση ενός γνωρίσματος, όπως για παράδειγμα το χρώμα ματιών. Οι διαφορετικές πιθανές «τοποθετήσεις» για ένα γνώρισμα (π.χ., μπλε, καφέ, μαύρο) καλούνται αλληλόμορφα γονίδια(alleles). Κάθε γονίδιο βρίσκεται σε έναν ιδιαίτερο γεωμετρικό τόπο (locus-θέση) στο χρωμόσωμα.

Πολλοί οργανισμοί έχουν πολλά χρωμοσώματα σε κάθε κύτταρο. Η πλήρης συλλογή του γενετικού υλικού (όλα τα χρωμοσώματα συνολικά) καλούνται γονιδίωμα του οργανισμού (genome). Ο όρος γενότυπος (genotype) αναφέρεται στο ιδιαίτερο σύνολο από γονίδια (genes) που περιλαμβάνονται σε ένα γονιδίωμα (genome). Δύο άτομα που έχουν τα ίδια γονιδιώματα λέγεται ότι έχουν τον ίδιο γενότυπο. Ο γενότυπος μετατρέπεται στο φαινότυπο του οργανισμού (phenotype), δηλαδή καθορίζει τα φυσικά και διανοητικά χαρακτηριστικά, όπως το χρώμα ματιών, το ύψος, το μέγεθος εγκεφάλου, και τη νοημοσύνη. Οι οργανισμοί των οποίων τα χρωμοσώματα παρατάσσονται ανά ζευγάρια (pair) ονομάζονται diploid ενώ οργανισμοί των οποίων τα χρωμοσώματα είναι αταίριαστα ονομάζονται haploid.

Κατά τη διάρκεια αναπαραγωγής, εμφανίζεται η λεγόμενη διασταύρωση (crossover): τα γονίδια από κάθε γονέα, ανταλλάσσονται μεταξύ τους σε ένα ζευγάρι χρωμοσωμάτων για να διαμορφώσουν ένα ενιαίο χρωμόσωμα (gamete), και έπειτα οι gametes από το αντίστοιχο ζευγάρι των δύο γονέων για να δημιουργήσουν ένα πλήρες σύνολο diploid χρωμοσωμάτων. Στην haploid αναπαραγωγή, τα γονίδια ανταλλάσσονται μεταξύ των μονόκλωνων χρωμοσωμάτων των δύο γονέων.

Ο απόγονος υπόκειται στη μεταλλαγή (mutation), στην οποία ενιαίες νουκλεοτίδες (στοιχειώδη κομμάτια DNA) αλλάζουν όταν μεταφέρονται από το γονέα στον απόγονο. Οι αλλαγές συχνά προκύπτουν ως αποτέλεσμα λάθους κατά την αντιγραφή. Η ικανότητα ή η υγεία ενός οργανισμού (fitness) ορίζεται είτε από την πιθανότητα επιβίωσης του οργανισμού ώστε να μπορέσει να αναπαραγάγει (βιωσιμότητα - viability), είτε συναρτήσει του αριθμού των απογόνου που έχει ένας οργανισμός (γονιμότητα - fertility).

4.4. Πως λειτουργούν

Ο τρόπος λειτουργίας των Γενετικών Αλγορίθμων είναι εμπνευσμένος από την βιολογία όπως έχω ήδη αναφέρει. Χρησιμοποιεί την ιδέα της εξέλιξης μέσω γενετικής μετάλλαξης, φυσικής επιλογής και διασταύρωσης.

Στην πράξη ο αλγόριθμος ξεκινά μ' ένα σύνολο λύσεων - ονομάζονται γονιδιώματα, δανειζόμενες το όνομά τους από τη βιολογία- οι οποίες συνιστούν τον "πληθυσμό". Κατόπιν ζητείται από τον υπολογιστή να δημιουργήσει μια σειρά τυχαίων ανασυνδυασμών και μεταλλάξεων των "γονιδιωμάτων".

Οι πιο ικανές λύσεις για ένα συγκεκριμένο πρόβλημα συνεχίζουν να εξελίσσονται και ανασυνδυάζονται τυχαία, μέχρις ότου "επιβιώσουν" οι καλύτερες. Συνήθως, όσο περισσότερες γενιές περνούν τόσο καλύτερες λύσεις βρίσκονται, μπορεί όμως ο αλγόριθμος να βρεθεί σε σημείο του πεδίου των λύσεων από όπου και δεν μπορεί να προχωρήσει λόγο του ότι βρίσκεται σε τοπικό μέγιστο. Για το λόγο αυτό έχουν υπάρχουν διαφορετικές εκδοχές του αλγόριθμου ανάλογα με τη μορφή του προβλήματος.

4.5. Περιγραφή τρόπου υλοποίησης γενετικών αλγορίθμων

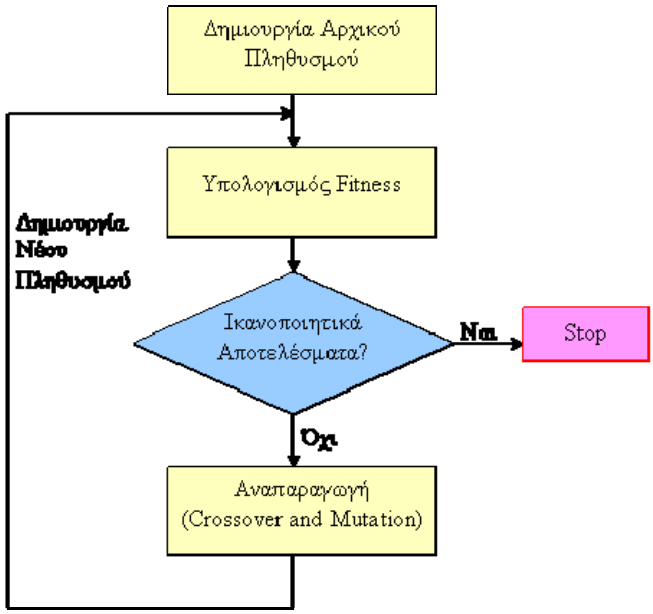
Οι Γενετικοί Αλγόριθμοι είναι αρκετά απλοί στην υλοποίησή τους. Οι τιμές για τις παραμέτρους του συστήματος πρέπει να κωδικοποιούνται με τρόπο ώστε να αναπαρασταθούν από μια μεταβλητή που περιέχει σειρά χαρακτήρων ή δυαδικών ψηφίων (0/1). Αυτή η μεταβλητή μιμείται το γενετικό κώδικα (γονιδίωμα) που υπάρχει στους ζωντανούς οργανισμούς.

Αρχικά, ο Γενετικός Αλγόριθμος παράγει πολλαπλά αντίγραφα της μεταβλητής/γενετικού κώδικα, συνήθως με τυχαίες τιμές, δημιουργώντας ένα πληθυσμό λύσεων. Κάθε λύση (τιμές για τις παραμέτρους του συστήματος) δοκιμάζεται για το πόσο κοντά φέρνει την αντίδραση του συστήματος στην επιθυμητή, μέσω μιας συνάρτησης που δίνει το μέτρο ικανότητας της λύσης και η οποία ονομάζεται συνάρτηση ικανότητας, όπως ανάφερα και πριν.

Οι λύσεις που βρίσκονται πιο κοντά στην επιθυμητή, σε σχέση με τις άλλες, σύμφωνα με το μέτρο που μας δίνει η συνάρτηση ικανότητας, αναπαράγονται στην επόμενη γενιά λύσεων και λαμβάνουν μια τυχαία μετάλλαξη. Επαναλαμβάνοντας αυτή τη διαδικασία για αρκετές γενιές, οι τυχαίες μεταλλάξεις σε συνδυασμό με την επιβίωση και αναπαραγωγή των γονιδίωματων/λύσεων που πλησιάζουν καλύτερα το επιθυμητό αποτέλεσμα θα παράγουν ένα γονίδιο/λύση που θα περιέχει τις τιμές για τις παραμέτρους που ικανοποιούν όσο καλύτερα γίνεται την συνάρτηση ικανότητας.

Όταν υλοποιούμε ένα παράδειγμα γενετικού αλγορίθμου ακολουθούμε τα εξής βήματα:

1. Αρχικοποιούμε τον πληθυσμό (population),
2. Υπολογίζουμε το fitness για κάθε γονιδίωμα του πληθυσμού,
3. Αναπαράγουμε επιλεγμένα γονιδιώματα για να σχηματίσουμε καινούριο πληθυσμό,
4. Γίνεται επανάληψη από το βήμα 2 μέχρι να ικανοποιείται μια συνθήκη τερματισμού.



Σχήμα 4.1: Βήματα Γενετικού Αλγορίθμου

4.5.1. Δημιουργία Αρχικού Πληθυσμού

Το πρώτο στάδιο των γενετικών αλγορίθμων, όπως φαίνεται και από το σχήμα 4.1, είναι η δημιουργία αρχικού πληθυσμού. Για να δημιουργηθεί ο αρχικός πληθυσμός παράγονται τυχαία πολλές μεμονωμένες λύσεις οι οποίες και τον διαμορφώνουν. Δημιουργείται δηλαδή ένα πλήθος εμπλεκόμενων λύσεων (χρωματοσώματα), στις οποίες θα εφαρμοστούν οι τεχνικές εξέλιξης (παρουσιάζονται στη συνέχεια).

Το μέγεθος του πληθυσμού εξαρτάται από τη φύση του προβλήματος (συνήθως περιέχει εκατοντάδες ή χιλιάδες πιθανές λύσεις). Κάθε χρωματόσωμα αποτελεί ένα σύνολο πιθανών λύσεων και είναι συνήθως ένα σύνολο δυαδικών αριθμών. Παραδοσιακά, ο πληθυσμός παράγεται τυχαία, καλύπτοντας όλο το διάστημα αναζήτησης αλλά περιστασιακά οι λύσεις μπορούν να αναζητηθούν στις περιοχές όπου είναι πιθανότερο να βρεθούν οι λύσεις οι οποίες είναι βέλτιστες.

4.5.2. Αξιολόγηση – Fitness

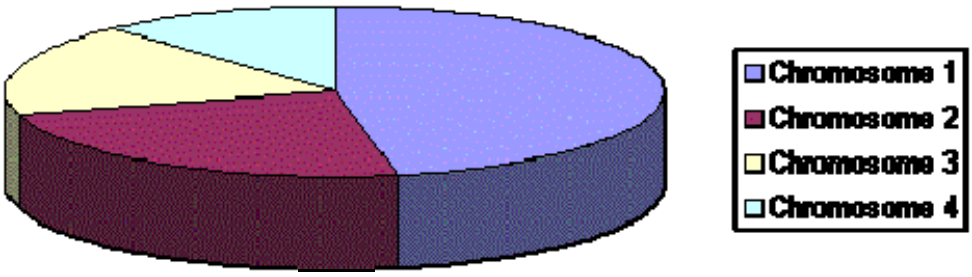
Η αξιολόγηση η οποία είναι το επόμενο βήμα γίνεται μέσω μιας συνάρτησης της fitness function (συνάρτηση ποιότητας), η οποία αξιολογεί την τρέχουσα λύση του προβλήματος. Δηλαδή για κάθε χρωμόσωμα x του πληθυσμού, υπολογίζεται η καταλληλότητα του (f(x)). Η συνάρτηση αυτή μας δίνει σαν αποτέλεσμα το κατά πόσο η υπό εξέταση λύση ταιριάζει με το επιθυμητό αποτέλεσμα (τον στόχο). Όταν η λύση αποκλίνει από το επιθυμητό αποτέλεσμα μπορούμε να θέτουμε μια ποινή (penalty), η οποία θα μειώνει την καταλληλότητα του συγκεκριμένου χρωμοσώματος. Στις περισσότερες εφαρμογές, το fitness value αντιπροσωπεύει το πόσο κοντά ή πόσο μακριά είμαστε από την λύση.

4.5.3. Επιλογή – Selection

Αφού αξιολογηθεί ο υπάρχον πληθυσμός, γίνεται η διαδικασία της επιλεκτικής αναπαραγωγής. Δηλαδή από τον υπάρχον πληθυσμό επιλέγουμε τα πιο υγιή χρωματοσώματα, αυτά με το καλύτερο fitness, σύμφωνα με κάποιο μηχανισμό επιλογής (παρουσιάζονται κάποιοι στην συνέχεια). Τα λιγότερο υγιή χρωμοσώματα θα εκλείψουν ενώ τα υπόλοιπα θα παρουσιαστούν και στην συνέχεια και πιθανών πάνω από μια φορά (ο αριθμός χρωμοσωμάτων μπορεί να μεταβάλλεται ή να παραμένει σταθερός ανάλογα με την επιθυμία του χρήστη). Όσο καλύτερη είναι η ποιότητα ενός ατόμου, τόσο μεγαλύτερη είναι η πιθανότητα να επιλεγεί περισσότερες φορές σαν γονέας για την αναπαραγωγή απογόνων.

- **Επιλογή βάση τροχού ρουλέτας (Roulette wheel selection)**

Ο μηχανισμός επιλογής βάση τροχού ρουλέτας είναι από τους πιο συχνά χρησιμοποιούμενους μηχανισμούς. Ο συγκεκριμένος μηχανισμός είναι ο μηχανισμός που χρησιμοποιείται και στην παρούσα εργασία. Η ιδέα του τροχού ρουλέτας, είναι ότι δίνεται ευκαιρία σε κάθε χρωματόσωμα να επιλεχθεί ως γονέας ανάλογα με το fitness του (την ποιότητά του). Ονομάζεται επιλογή βάση του τροχού ρουλέτας επειδή μπορούμε να το δούμε ως τον τροχό μιας ρουλέτας όπου το μέγεθος του κάθε κομματιού (slot) είναι ανάλογο με την ποιότητά του κάθε χρωματοσώματος. Είναι εμφανές ότι τα χρωματοσώματα με την καλύτερη ποιότητα (με το πιο μεγάλο κομμάτι στη ρουλέτα), έχουν περισσότερες πιθανότητες να επιλεγούν. Αυτό παρουσιάζεται διαγραμματικά στο σχήμα που φαίνεται πιο κάτω, όπου η λύση (χρωματόσωμα 1) είναι το πιο υγιές, έχει το μεγαλύτερο ποσοστό στο τροχό και επομένως έχει την μεγαλύτερη πιθανότητα να επιλεγεί σε κάθε γύρισμα του τροχού.



Σχήμα 4.2: Επιλογή βάση του τροχού ρουλέτας

Ο αλγόριθμος για τον τροχό της ρουλέτας παρουσιάζεται πιο κάτω:

1. Μετά που έχουμε υπολογίσει το fitness value για κάθε χρωμόσωμα (eval(u_i) από i = 0 μέχρι i = pop_size), αθροίζουμε όλα τα fitness value για να βρούμε το συνολικό fitness όλων των χρωμοσωμάτων το οποίο συμβολίζεται ως F.

   $F = \sum_{i=1}^{pop\_size} eval(u_i)$, όπου pop_size είναι το μέγεθος του πληθυσμού

2. Ακολούθως υπολογίζουμε την πιθανότητα επιλογής κάθε χρωμοσώματος p_i (probability of selection) για κάθε χρωμόσωμα:

   $$p_i = \frac{eval(u_i)}{F}$$

3. Και στο τέλος υπολογίζουμε το cumulative probability q_i για κάθε χρωμόσωμα u_i από τον τύπο:

   $$q_i = \sum_{j=1}^{i} p_j$$

Με την ολοκλήρωση των πιο πάνω διαδικασιών, γυρίζουμε τον τροχό της ρουλέτας τόσες φορές όσες είναι ο πληθυσμός μας, και κάθε φορά επιλέγουμε ένα χρωμόσωμα για τον νέο πληθυσμό με τον εξής τρόπο:

1. Δημιουργούμε ένα τυχαίο αριθμό r μεταξύ 0 και 1.
2. Αν το $r < q_1$ τότε επιλέγουμε το 1ο χρωματόσωμα (u_1), αλλιώς αν ισχύει $q_{i-1} < r \leq q_i$ επιλέγουμε το i-ιοστό χρωματόσωμα u_i ($2 \leq i \leq pop\_size$).

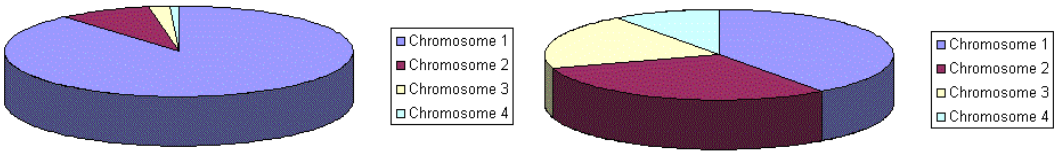
- **Επιλογή με βαθμολόγηση (Ranking selection)**

Όταν υπάρχουν μεγάλες διαφορές μεταξύ των fitness value (τιμών ποιότητας), ο προηγούμενος μηχανισμός επιλογής μπορεί να παρουσιάσει προβλήματα. Για παράδειγμα αν το καλύτερο individual έχει ποιότητα 90%, τότε τα υπόλοιπα χρωματοσώματα θα έχουν πολύ λίγες πιθανότητες να επιλεχθούν.

Η επιλογή με βαθμολόγηση, ταξινομεί αρχικά τον πληθυσμό και στην συνέχεια δίνεται σε κάθε individual ένα αύξων αριθμό ο οποίος καθορίζεται από το fitness value. Το χειρότερο individual θα έχει fitness 1, το 2ο χειρότερο 2 κ.τ.λ. Το καλύτερο χρωματόσωμα θα έχει fitness N, όπου N είναι ο αριθμός των individuals στον πληθυσμό. Στη συνέχεια μια ειδική συνάρτηση καθορίζει πόσες φορές θα επιλεγεί ο καθένας από τους γονείς. Με τον τρόπο αυτό, σε αντίθεση με της ρουλέτας, ακόμα και τα χειρότερα individuals έχουν πιο σημαντική πιθανότητα να επιλεχθούν από ότι στην επιλογή ρουλέτας.

Παρόλα αυτά η μέθοδος αυτή μπορεί να οδηγήσει σε καθυστερημένη σύγκληση (αργεί να ολοκληρωθεί) λόγω του γεγονότος ότι τα καλύτερα χρωμοσώματα με πολύ ψηλό fitness δεν διαφέρουν τόσο πολύ από αυτά με χαμηλότερο.

Στο πιο κάτω σχήμα παρουσιάζεται η μετατροπή της ποιότητας κάθε λύσης σε σύστημα επιλογής με βαθμολόγηση. Στο πρώτο σχήμα φαίνεται η πραγματική ποιότητα του κάθε χρωματοσώματος ενώ στο δεύτερο η μετατρεπόμενη τιμή σε επιλογή με βαθμολόγηση. Είναι εύκολο να διακρίνει κανείς ότι η πιθανότητα επιλογής του χρωματοσώματος 1 μειώνεται ενώ του 4 αυξάνεται. Η διακύμανση αυτή αποκρύπτει την πραγματική διαφορά ποιότητας μεταξύ των χρωματοσωμάτων. Αυτό αποτελεί και το μειονέκτημα της μεθόδου αυτής.



Σχήμα 4.3: Μετατροπή πραγματικής ποιότητας (1^{ο} σχήμα) σε σύστημα επιλογής βαθμολόγησης (2^{ο} σχήμα)

- **Επιλογή με τουρνουά (tournament selection)**

Για την μέθοδο αυτή υπάρχουν διάφορες παραλλαγές. Το κοινό σε όλες τις παραλλαγές είναι ότι για να επιλεγεί ένας γονέας, διαχωρίζεται κάποια ομάδα ατόμων και από αυτά επιλέγεται το χρωματόσωμα (individual) με το μεγαλύτερο fitness για την αναπαραγωγή. Αυτό επαναλαμβάνεται N φορές για ένα πληθυσμό μεγέθους N. Μερικές παραλλαγές παρουσιάζονται πιο κάτω.

1. Επιλέγουμε δύο individuals τυχαία και επιλέγουμε το individual με το μεγαλύτερο fitness για να γίνει γονέας.
2. Παρόμοια με την προηγούμενη παραλλαγή, απλά επιλέγουμε περισσότερα από δύο individuals κάθε φορά και επιλέγουμε για αναπαραγωγή αυτό με το μεγαλύτερο fitness, με μια καθορισμένη πιθανότητα.

3. Επιλέγουμε ένα ζευγάρι individuals τυχαία. Δημιουργούμε ένα τυχαίο αριθμό R μεταξύ του 0 και του 1. Αν το R < r επιλέγεται για αναπαραγωγή το 1ο individual. Αν το R >= r επιλέγεται για αναπαραγωγή το 2ο individual. Το r είναι μια παράμετρος στη μέθοδο αυτή.

4.5.4. Αναπαραγωγή – Reproduction

Όταν γίνει η επιλογή των γονέων, εφαρμόζοντας μια μέθοδο από τις πιο πάνω, ακολουθεί η αναπαραγωγή κατά την οποία μπορούν να εφαρμοστούν δύο διαδικασίες: η διασταύρωση (**crossover**) και η μετάλλαξη (**mutation**). Οι διαδικασίες αυτές αναλύονται πιο κάτω. Γενικά όταν έχουμε Ν γονείς θα παραχθούν και Ν παιδιά.

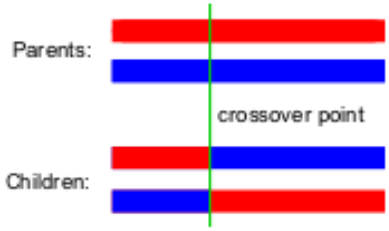
4.5.5. Διασταύρωση – Crossover

Η διασταύρωση είναι ο τελεστής που ανά-συνδυάζει τους γονότυπους δύο γονέων, ανταλλάσσοντας τμήματα και δημιουργώντας νέο γονότυπο. Χαρακτηρίζεται ως η βασικότερη γενετική διαδικασία.

Αρχικά ομαδοποιούμε τα χρωματοσώματά μας σε δυάδες με τυχαίο τρόπο και για κάθε δυάδα καθορίζουμε αν θα γίνει η διασταύρωση ή όχι. Ορίζουμε ένα ποσοστό διασταύρωσης και δημιουργούμε ένα τυχαίο αριθμό για κάθε ζευγάρι. Έτσι, ανάλογα με το αν ο αριθμός του ζευγαριού ξεπερνά ή όχι το ποσοστό, καθορίζουμε αν θα γίνει η διασταύρωση ή όχι. Με τον τρόπο αυτό κάποια ζευγάρια χρωμοσωμάτων επιλέγονται για διασταύρωση και άλλα όχι. Στην συνέχεια για τα ζευγάρια που επιλέχθηκαν, επιλέγουμε τυχαία τα σημεία στα οποία θα γίνει η διασταύρωση (crossover points). Ανάλογα με το πόσα είναι τα crossover points έχουμε τις ακόλουθες μεθόδους.

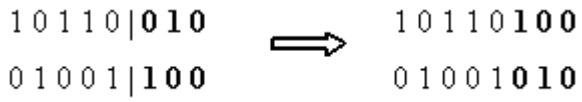
- **Διασταύρωση ενός σημείου (Single point crossover ή one point crossover)**

Σε αυτή την παραλλαγή διασταύρωσης, επιλέγεται τυχαία ένα σημείο διασταύρωσης και ο κάθε γονέας χωρίζεται σε δύο τμήματα. Στην συνέχεια ανταλλάσσονται τα δύο τμήματα μεταξύ των δύο γονέων όπως φαίνεται στο πιο παράδειγμα που ακολουθεί.



Σχήμα 4.4: One point crossover

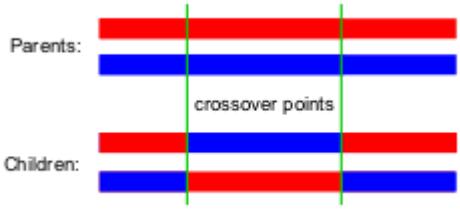
Αυτή είναι και η πιο βασική μέθοδος διασταύρωσης όπως περιγράφουν και οι Holland, Goldberg και άλλοι στο [23]. Για binary αριθμούς η διασταύρωση ενός σημείου θα γινόταν ως εξής:



Σχήμα 4.5: One point crossover για binary αριθμούς

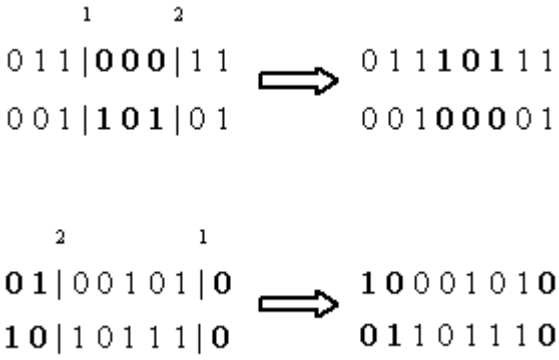
- **Διασταύρωση δύο σημείων (Two point crossover)**

Με αυτή τη μέθοδο, επιλέγονται δύο σημεία κοπής. Οπόταν ο κάθε γονέας χωρίζεται σε τρία τμήματα. Μετά ανταλλάσσεται το ένα από τα τρία τμήματα μεταξύ του ζευγαριού όπως στο πιο ακόλουθο παράδειγμα.



Σχήμα 4.6: Two point crossover

Για binary αριθμούς η διασταύρωση ενός σημείου θα γινόταν ως εξής:



Σχήμα 4.7: Two point crossover για binary αριθμούς

- **Διασταύρωση πολλαπλών σημείων (Multi point crossover)**

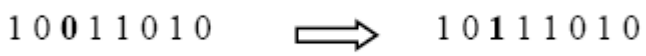
Η μέθοδος αυτή αποτελεί μια άλλη παραλλαγή του crossover, όπου ο κάθε γονέας χωρίζεται σε N σημεία κοπής τα οποία επιλέγονται τυχαία. Δηλαδή ο κάθε γονέας χωρίζεται σε N+1 τμήματα από τα οποία κάποια ανταλλάσσονται μεταξύ των 2 γονέων.

- **Ομοιόμορφη διασταύρωση (Uniform crossover)**

Αυτή η διασταύρωση περιγράφτηκε από τον Syswerda στο [24]. Στην ομοιόμορφη διασταύρωση για κάθε bit position στο χρωματόσωμα αποφασίζουμε τυχαία αν θα γίνει ή όχι η ανταλλαγή των bits μεταξύ των δύο γονέων. Έτσι σύμφωνα με την μέθοδο αυτή κάθε σύμβολο του γονότυπου του απογόνου λαμβάνεται με ίση πιθανότητα από τον ένα ή τον άλλο γονέα.

4.5.6. Μετάλλαξη - Mutation

Με τον όρο mutation εννοούμε μια τυχαία αλλαγή των bits που παρουσιάζεται σε κάθε παραγωγή. Δηλαδή με την μέθοδο αυτή αλλάζουμε κάποια από τα bits του γονέα για να δημιουργήσουμε ένα απόγονο. Η διαδικασία αυτή γίνεται από bit-σε-bit σε ολόκληρο τον υπάρχον πληθυσμό και σκοπός της είναι να καλύψει την αδυναμία του crossover να δημιουργήσει πληροφορίες που δεν περιέχονται στον πληθυσμό. Με άλλα λόγια επιτυγχάνουμε την μη πόλωση των λύσεων σε μια μόνο συγκεκριμένη κατεύθυνση.



Σχήμα 4.8: Μετάλλαξη στο 3^{ο} γονίδιο

Ο παραπάνω είναι ο ορισμός της μετάλλαξης για δυαδικούς αριθμούς. Μετάλλαξη μπορεί να γίνει και με ακέραιες κωδικοποιήσεις, αλλάζοντας π.χ. τον αριθμό που αναπαριστάτε προσθέτοντας του ή αφαιρόντας του ένα πολύ μικρό αριθμό. Επίσης μπορούμε να σκεφτούμε και άλλων ειδών μεταλλάξεων, δεν υπάρχει περιορισμός.

4.6. Λόγοι που οδήγησαν στους γενετικούς αλγορίθμους

Οι γενετικοί αλγόριθμοι αναπτύχθηκαν για ένα μεγάλο αριθμό από λόγους παρόλο που υπήρχε μια σχετική δυσπιστία στην αρχή για το αν αυτό το μοντέλο αλγορίθμου θα λειτουργούσε καλά. Παρόλα αυτά αποδείχθηκε ότι οι γενετικοί αλγόριθμοι υπερνικούσαν τα μειονεκτήματα των κλασικών μεθόδων αναζήτησης και βελτιστοποίησης. Παρακάτω παρουσιάζω τους λόγους και τα πλεονεκτήματα των γενετικών αλγορίθμων.

4.6.1. Πλεονεκτήματα γενετικών αλγορίθμων

Η χρήση των Γ.Α. σε διάφορες εφαρμογές είναι ελκυστική για αρκετούς λόγους. Τα πλεονεκτήματα που παρουσιάζουν περιγράφονται την συνέχεια εν συντομία.

1. Μπορούν να λύσουν δύσκολα προβλήματα γρήγορα και αξιόπιστα.

Ένας από τους πιο σημαντικούς λόγους χρήσης των Γ.Α. είναι η μεγάλη αποδοτικότητα που διαθέτουν. Τόσο η θεωρία, όσο και η πράξη έχουν δείξει ότι προβλήματα που έχουν πολλές και δύσκολα προσδιορισμένες λύσεις μπορούν να αντιμετωπιστούν καλύτερα από Γ.Α. Είναι δε αξιοσημείωτο, ότι συναρτήσεις που παρουσιάζουν μεγάλες διακυμάνσεις και καθιστούν άλλες μεθόδους ανεπαρκείς στην εύρεση των ακρότατων τους, με τους Γ.Α. δεν αποτελούν ιδιαίτερο πρόβλημα.

2. Μπορούν εύκολα να συνεργαστούν με υπάρχοντα μοντέλα και συστήματα.

Οι Γ.Α. προσφέρουν το σημαντικό πλεονέκτημα της άμεσης χρήσης τους (με προσθετικό τρόπο) στα συστήματα που χρησιμοποιούνται σήμερα, μη απαιτώντας την επανασχεδίασή τους. Μπορούν εύκολα να συνεργαστούν με τον υπάρχοντα κώδικα, χωρίς ιδιαίτερη προσπάθεια και τροποποιήσεις. Αυτό συμβαίνει, γιατί οι Γ.Α. χρησιμοποιούν μόνο πληροφορίες της διαδικασίας ή της συνάρτησης που πρόκειται να βελτιστοποιήσουν, δίχως να τους ενδιαφέρει άμεσα ο ρόλος της συνάρτησης μέσα στο σύστημα.

3. Είναι εύκολα επεκτάσιμοι και εξελίξιμοι.

Οι Γ.Α. μπορούν εύκολα να αφομοιώσουν αλλαγές, επεκτάσεις και μετεξελίξεις, που μπορεί να χρειαστούν. Σε πολλές εφαρμογές, έχουν αναφερθεί λειτουργίες των Γ.Α. που δεν είναι δανεισμένες από τη φύση ή που έχουν υποστεί σημαντικές αλλαγές, πάντα προς όφελος της απόδοσης. Παραλλαγές στο βασικό σχήμα δεν είναι απλά αναγκαίες, αλλά σε ορισμένες περιπτώσεις επιβάλλονται.

4. Μπορούν να χρησιμοποιηθούν σε υβριδικές μορφές με άλλες μεθόδους.

Αν και η ισχύς των Γ.Α. είναι μεγάλη, σε μερικές ειδικές περιπτώσεις προβλημάτων, όπου άλλες μέθοδοι συμβαίνει να έχουν πολύ υψηλή αποδοτικότητα, λόγω εξειδίκευσης, υπάρχει η δυνατότητα χρησιμοποίησης ενός υβριδικού σχήματος Γ.Α. σε συνδυασμό με άλλη μέθοδο. Αυτό μπορεί να συμβεί αφού οι Γ.Α. διαθέτουν μεγάλη ευελιξία.

5. Δεν απαιτούν περιορισμούς στις συναρτήσεις που επεξεργάζονται.

Ο κύριος λόγος που καθιστά τις παραδοσιακές μεθόδους δύσκαμπτες και ακατάλληλες για πολλά προβλήματα είναι περιορισμοί όπως η απαίτησή τους για ύπαρξη παραγώγων, ύπαρξη συνέχειας, κ.τ.λ. Τέτοιου είδους περιορισμοί αφήνουν αδιάφορους τους Γ.Α. πράγμα που τους κάνει κατάλληλους για μεγάλο φάσμα προβλημάτων.

6. Δεν ενδιαφέρει η σημασία της υπό εξέταση πληροφορίας.

Η μόνη "επικοινωνία" του Γ.Α. με το περιβάλλον του είναι η συνάρτηση αξιολόγησης (fitness function). Αυτή η ιδιότητα εγγυάται την επιτυχία του αλγορίθμου ανεξάρτητα από την σημασία του προβλήματος. Βέβαια, αυτό δεν σημαίνει ότι δεν υπάρχουν άλυτα προβλήματα για τους Γ.Α. Όπου όμως δεν τα καταφέρνουν, η αιτία πιθανών να είναι η φύση του χώρου που ερευνούν και όχι το πληροφοριακό περιεχόμενο του προβλήματος.

7. Έχουν από τη φύση τους το στοιχείο του παραλληλισμού.

Οι Γ.Α. σε κάθε τους βήμα επεξεργάζονται μεγάλες ποσότητες πληροφορίας, αφού κάθε άτομο θεωρείται αντιπρόσωπος πολλών άλλων. Έχει υπολογιστεί ότι 10 άτομα αντιπροσωπεύουν περίπου 1000. Για τον λόγο αυτό, μπορούν να καλύψουν με αποδοτικό ψάξιμο μεγάλους χώρους σε μικρούς χρόνους.

8. Είναι η μόνη μέθοδος που κάνει ταυτόχρονα εξερεύνηση του χώρου αναζήτησης, και εκμετάλλευση της ήδη επεξεργασμένης πληροφορίας.

Ο συνδυασμός αυτός σπάνια συναντάτε σε οποιαδήποτε άλλη μέθοδο. Με το τυχαίο ψάξιμο γίνεται καλή εξερεύνηση του χώρου, αλλά δεν γίνεται εκμετάλλευση της πληροφορίας. Αντίθετα, με το hill-climbing γίνεται καλή εκμετάλλευση της πληροφορίας, αλλά όχι καλή εξερεύνηση. Συνήθως τα δύο αυτά χαρακτηριστικά είναι ανταγωνιστικά και το επιθυμητό είναι να συνυπάρχουν και τα δύο προς όφελος της όλης διαδικασίας. Οι Γ.Α. επιτυγχάνουν το βέλτιστο συνδυασμό εξερεύνησης και εκμετάλλευσης, πράγμα που τους κάνει ιδιαίτερα αποδοτικούς και ελκυστικούς.

9. Επιδέχονται παράλληλη υλοποίηση.

Οι Γ.Α. μπορούν να εκμεταλλευτούν τα πλεονεκτήματα των παράλληλων μηχανών, αφού λόγω της φύσης τους, εύκολα μπορούν να δεχτούν παράλληλη υλοποίηση. Το χαρακτηριστικό αυτό αυξάνει ακόμη περισσότερο την απόδοσή τους, ενώ σπάνια συναντάτε σε ανταγωνιστικές μεθόδους.

10. Εφαρμόζονται σε πολύ περισσότερα πεδία από κάθε άλλη μέθοδο.

Το χαρακτηριστικό που τους εξασφαλίζει αυτό το πλεονέκτημα είναι η ελευθερία επιλογής των κριτηρίων που καθορίζουν την επιλογή μέσα στο τεχνικό περιβάλλον. Έτσι, οι Γ.Α. μπορούν να χρησιμοποιηθούν στην οικονομία, στο σχεδιασμό μηχανών, στην επίλυση μαθηματικών εξισώσεων, στην εκπαίδευση των Νευρωνικών Δικτύων και σε πολλούς άλλους τομείς.

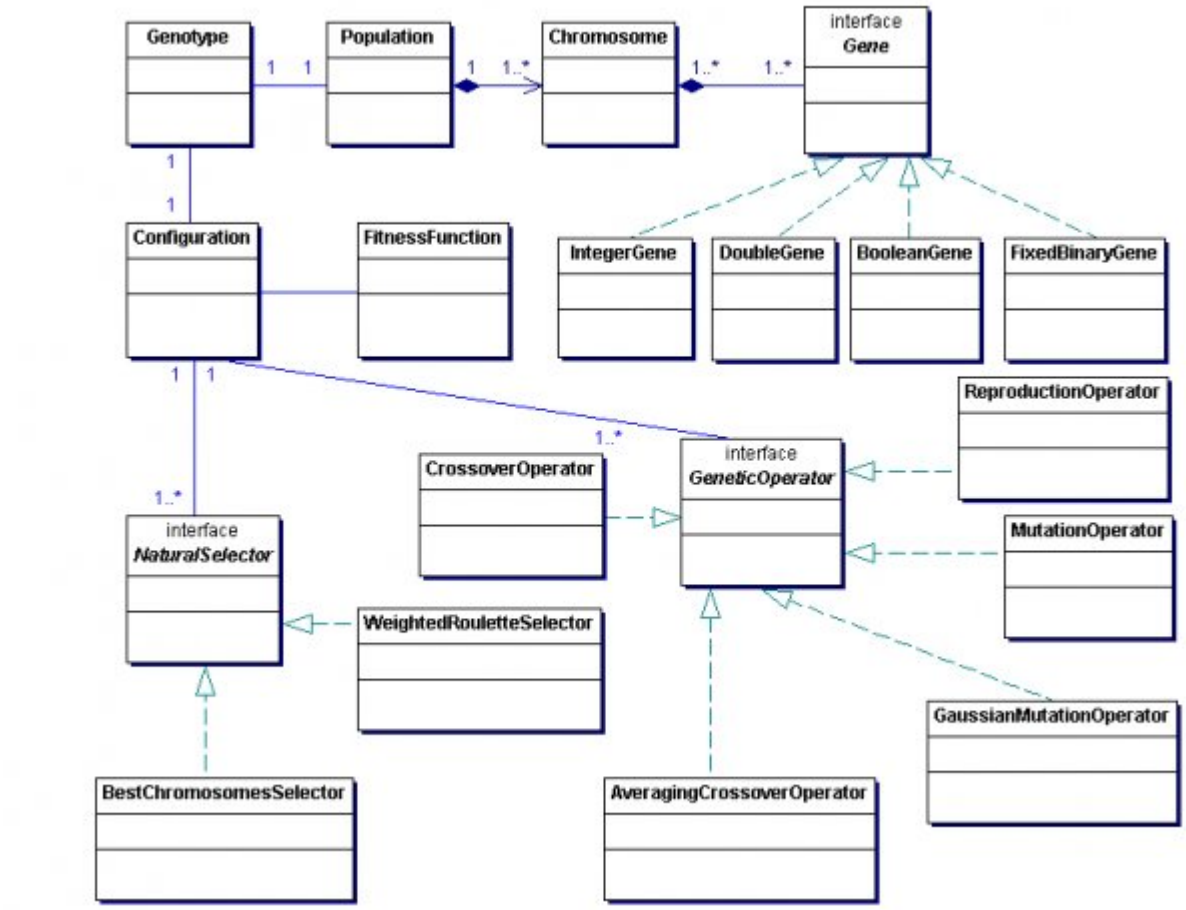
4.7. JGAP: Βιβλιοθήκη υλοποίησης γενετικών αλγορίθμων

Η δημιουργία βιβλιοθήκης η οποία θα υλοποιεί τις αρχές των γενετικών αλγορίθμων δεν αποτελεί αυτό καθαυτό πρόκληση. Όπως έχει ήδη προαναφερθεί οι Γ.Α. είναι αυστηρά ορισμένοι με γνωρίσματα που τους καθιστούν τόσο επεκτάσιμους όσο και με μεγάλη δυνατότητα για παραμετροποίηση. Αυτή η τελευταία δυνατότητα είναι που επιφέρει και την πραγματική πρόκληση κατά την δημιουργία βιβλιοθήκης Γ.Α.

Στην ατομική μεταπτυχιακή εργασία του Κύπρου Οικονομίδη στο [5] χρησιμοποιήθηκαν γενετικοί αλγόριθμοι. Σκοπός της μεταπτυχιακή του εργασίας δεν ήταν να ξανά-ανακαλύψει τον τροχό! Έτσι η δημιουργία μιας ακόμα βιβλιοθήκης Γ.Α. θα ήταν άτοπη, αφού υπάρχουν ήδη έτοιμες και αξιόλογες προτάσεις. Στο διαδίκτυο μπορεί κανείς να βρει αρκετές διαφορετικές υλοποιήσεις, άλλες πιο ευέλικτες και άλλες πιο δύσκαμπτες. Κάποιες από αυτές διατίθενται δωρεάν ενώ άλλες όχι. Ιδανικότερη βιβλιοθήκη για την εργασία του αποδείχτηκε να είναι η JGAP [25]. Στην συνέχεια θα γίνει μια συνοπτική παρουσίαση της βιβλιοθήκης αυτής η οποία χρησιμοποιείται και στην παρούσα εργασία.

4.7.1. Τι είναι το JGAP

Η JGAP είναι μια βιβλιοθήκη υλοποίησης Γενετικών Αλγόριθμων και Γενετικού Προγραμματισμού που παρέχεται ως πλαίσιο (framework) της γλώσσας προγραμματισμού Java. Παρέχει βασικούς γενετικούς μηχανισμούς που μπορούν να χρησιμοποιηθούν εύκολα για να εφαρμόσουν τις εξελικτικές αρχές στην λύση προβλημάτων. Η δομή των πιο σημαντικών κλάσεων που υπάρχουν υλοποιημένες στην βιβλιοθήκη, παρουσιάζεται στο επόμενο σχήμα.



Σχήμα 4.9: Οι πιο σημαντικές κλάσεις του JGAP.

Η JGAP είναι εύχρηστη, ενώ επίσης είναι ιδιαίτερα επεκτάσιμη (αφού είναι οργανωμένη σε τεμάχια - modules) έτσι ώστε οι πιο έμπειροι χρήστες να μπορούν εύκολα να αντικαταστήσουν κομμάτια ή να προσαρμόσουν τα ήδη υπάρχοντα.

Κεφάλαιο 5

Παρουσίαση προηγούμενης εργασίας

5.1. Εισαγωγή.....85
5.2. Περιγραφή προηγούμενης εργασίας.....85
5.3. Παρόμοιες/παλιές εργασίες – Related work.....90

5.1. Εισαγωγή

Η προηγούμενη εργασία είχε τίτλο «Δημιουργία γράφου ροής δεδομένων και έλεγχος λογισμικού» [5]. Η μέθοδος που ακολουθήθηκε σ' αυτή την εργασία ήταν για έλεγχο κώδικα λογισμικού και περιλάμβανε την δημιουργία γράφου ροής δεδομένων (Data Flow Graphs) και χρήση γενετικών αλγορίθμων με απώτερο σκοπό την εκτίμηση της ποιότητας ενός προγράμματος. Η αρχιτεκτονική που χρησιμοποιήθηκε, είναι βασισμένη σε αυτή που δημιουργήθηκε στα πλαίσια άλλων Ατομικών Διπλωματικών Εργασιών και συγκεκριμένα στο [4]. Η δυναμική αυτή αρχιτεκτονική, επεκτάθηκε σε ορισμένα σημεία και υπέστη αλλαγές σε κάποια άλλα ώστε να προσαρμοστεί στα δεδομένα της εργασίας στο [5] και να κάνει το ολοκληρωμένο σύστημα ακόμα πιο ανοιχτό επιτρέποντας τον πειραματισμό σε διαφορετικούς συνδυασμούς τεχνολογιών. Μετά από αυτό, ακολούθησε η δημιουργία γενετικού αλγορίθμου και η προσαρμογή του ώστε να εκμεταλλεύεται την αρχιτεκτονική που δημιουργήθηκε και να παράγει δεδομένα εισόδου. Στόχος του γενετικού αλγορίθμου είναι η κάλυψη ενός κριτηρίου επάρκειας ελέγχου των Γράφων Ροής Δεδομένων. Το κριτήριο αυτό είναι το All-DU-Paths.

5.2. Περιγραφή προηγούμενης εργασίας

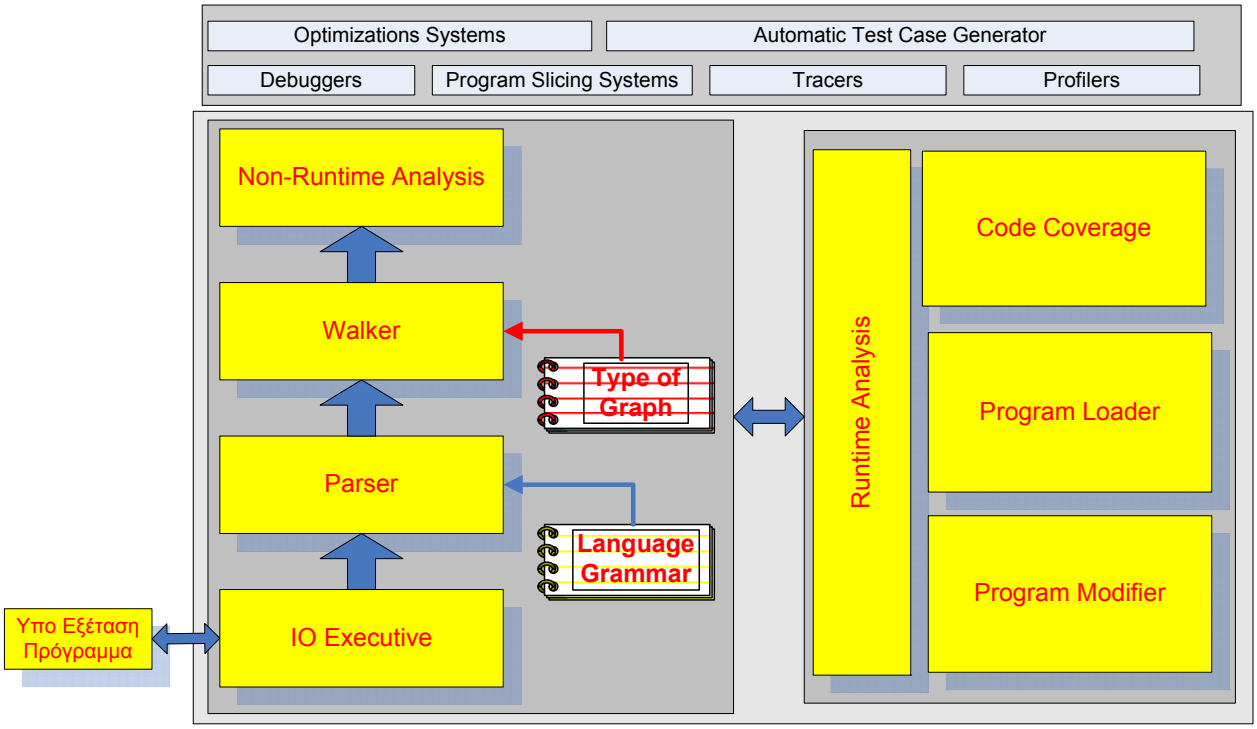
Στο κεφάλαιο αυτό, θα γίνει μια συνοπτική ανάλυση της προηγούμενης εργασία. Θα εστιάσω στο πως γίνεται η ανάλυση προγραμμάτων, ποια είναι η αρχιτεκτονική που χρησιμοποιήθηκε και στην παραγωγή γράφου ροής δεδομένων που έγιναν στην εργασία αυτή.

5.2.1. Αρχιτεκτονική αναλυτή προγραμμάτων

Η αρχιτεκτονική που χρησιμοποιήθηκε αποτελείται από διάφορα ιεραρχικά επίπεδα (layers) για την ανάλυση του κώδικα προγραμμάτων γραμμένο σε διάφορες γλώσσες προγραμματισμού. Η σχεδίαση της αρχιτεκτονικής αυτής είχε ως στόχους:

- Την κλιμακωσιμότητα, δηλαδή την δυνατότητα υποστήριξης μεγάλων προγραμμάτων
- Την εφαρμοσιμότητα, δηλαδή την δυνατότητα να εξετάζει τα χαρακτηριστικά γνωρίσματα των σύγχρονων γλωσσών προγραμματισμού και προγραμμάτων,
- Την δυνατότητα χρησιμοποίησης, δηλαδή να σχεδιαστεί ως πλατφόρμα επάνω στην οποία ποικίλα εργαλεία μπορούν να χτιστούν, κάθε ένα να μπορεί εξετάζει έναν ιδιαίτερο στόχο και ο προγραμματιστής μπορεί να εκτελέσει διάφορες λειτουργίες.

Η αρχιτεκτονική προτάθηκε αρχικά στο [26] και είναι αυτό που ονομάζουμε πολυεπίπεδη. Το κάθε επίπεδο ή στρώμα παρέχει τις υπηρεσίες του στα αμέσως ανώτερα στρώματα, ενώ για να λειτουργήσει χρησιμοποιεί τα δεδομένα ή τις υπηρεσίες που παράγονται από τα κατώτερα στρώματα. Μπορεί δηλαδή να παρομοιαστεί με την αρχιτεκτονική του OSI για τα δίκτυα επικοινωνιών. Το σύνολο της αρχιτεκτονικής, όπως παρουσιάζεται και στο πιο κάτω σχήμα, μπορεί να χρησιμοποιηθεί σαν ενδιάμεσο στάδιο για ανώτερες εφαρμογές όπως βελτιστοποίηση κώδικα, παραγωγή τυχαίων μεταβλητών εισόδου (test cases), προγράμματα αποσφαλμάτωσης (debuggers), κλπ. Στην συνέχεια παρουσιάζονται τα διαφορετικά επίπεδα της αρχιτεκτονικής.



Σχήμα 5.1: Αρχιτεκτονική προτεινόμενου αναλυτή προγραμμάτων

Το πρώτο επίπεδο ονομάζεται **IO Executive**. Είναι ένα σύνολο από υποπρογράμματα υπεύθυνα για την ανάγνωση και γραφή του πηγαίου κώδικα του προγράμματος το οποίο θα εξεταστεί.

Το επόμενο επίπεδο είναι ο **Parser**. Το επίπεδο παρέχει έναν διαπεραστή για την επιθυμητή γλώσσα προγραμματισμού. Έχει δυνατότητα διαπέρασης πηγαίου κώδικα ή κώδικα ψηφιολέξεων (bytecode), και εκτελεί ανάλυση των ονομάτων και τύπων του προγράμματος. Η γραμματική της γλώσσας μπορεί να φορτωθεί από ένα εξωτερικό αρχείο, δίνοντας έτσι την δυνατότητα επιλογής της γλώσσας του προγράμματος που θα ελεγχθεί. Στην συγκεκριμένη φάση χρησιμοποιούμε τη γραμματική της προγραμματιστικής γλώσσας Java.

Στην συνέχεια, τα δεδομένα που παράγει ο parser χρησιμοποιούνται από τον **Walker**. Ο Walker είναι το επίπεδο που είναι υπεύθυνο για "το περπάτημα" κάθε έκφρασης και (λαμβάνοντας υπόψη τυχόν περιορισμούς) την κατασκευή της γραφικής αναπαράστασης του πηγαίου κώδικα. Στο τέλος, αυτό που πετυχαίνει ο Walker, είναι να αποδώσει μια γραφική αναπαράσταση του προγράμματος δημιουργώντας τους κόμβους (vertices) και τις ακμές (edges) για τον γράφο ροής δεδομένων.

Το επίπεδο στατικής ανάλυσης ή ανάλυσης σε χρόνο μη εκτέλεσης (**non runtime analysis**), παρέχει τη διεπαφή για περαιτέρω ενότητες ανάλυσης, που συνδέονται με διεπαφή τύπου γραφικού περιβάλλοντος - GUI. Χρησιμοποιεί δομές δεδομένων και άλλες πληροφορίες που έχουν παραχθεί από τα προηγούμενα επίπεδα. Διαφορετικές δομές των στοιχείων χρησιμοποιούνται μαζί, για να παρέχουν τον τύπο ανάλυσης που μπορεί να χρειαστεί ο υπεύθυνος για τον έλεγχο (π.χ. γραφική παράσταση ροής δεδομένων, προσδιορισμός μεθόδων, εξαγωγή μονοπατιών, κλπ).

Η ενότητα ανάλυσης χρόνου εκτέλεσης (**runtime analysis**) χρησιμοποιείται για να υπερνικήσει τους περιορισμούς της στατικής ανάλυσης μέσω δυναμικής ανάλυσης. Αν και η χρήση της δυναμικής ανάλυσης μπορεί να επεκταθεί σε ποικίλους τομείς και εργασίες, στην εφαρμογή προσφέρει δύο σημαντικά χαρακτηριστικά γνωρίσματα, την προσομοίωση της εκτέλεσης (running simulation) και την κάλυψη κώδικα (code coverage). Συγκεκριμένα, με την παροχή ενός συνόλου εισόδων στο υπό εξέταση πρόγραμμα, η ανάλυση χρόνου εκτέλεσης μιμείται την εκτέλεση του προγράμματος και συγχρόνως είναι σε θέση να δείξει τον εκτελεσμένο / καλυμμένο (executed/covered) κώδικα και συνεπώς τον μη καλυμμένο (uncovered) κώδικα. Τα αποτελέσματα ισχύουν μόνο για το τρέχον τρέξιμο, δηλαδή για τις δεδομένες τιμές των μεταβλητών εισόδου. Σε αυτό το σημείο έγιναν αλλαγές από την προοηγούμενη δουλειά ώστε κατά την συμβολική εκτέλεση του προγράμματος, να χρησιμοποιούνται τα δεδομένα που υπάρχουν αποθηκευμένα σε ένα γράφο ροής δεδομένων.

5.2.2. Παραγωγή Γράφου Ροής Δεδομένων

Η παραγωγή του γράφου ροής δεδομένων ήταν ένα από τα σημαντικότερα κομμάτια της εργασίας στο [5]. Οι πληροφορίες σε ένα γράφο ροής δεδομένων δεν αποθηκεύονται μόνο σε κόμβους, όπως στην περίπτωση των γράφων ροής ελέγχου, αλλά και στις ακμές. Κατά την διαδικασία κατασκευής ο Walker χτίζει τον γράφο ανάλογα με το τις διαφορετικές δομές που συναντά στο πρόγραμμα.

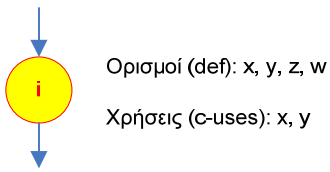
Λόγω της ιδιαιτερότητας που παρουσιάζει ο γράφος ροής δεδομένων έπρεπε να γίνουν ουσιαστικές αλλαγές στον Walker. Ο κυριότερος λόγος είναι ότι κατά το χτίσιμο του γράφου, ενδιαφερόμαστε περισσότερο για τις εμφανίσεις των μεταβλητών και για το πώς αυτές χρησιμοποιούνται. Στην συνέχεια θα παρουσιάσουμε το πώς συμπεριφέρεται ο Walker σε κάθε περίπτωση, ανάλογα με τον τύπο εντολής που συναντά κάθε φορά.

- **Εντολή Ανάθεσης**

Σε κάθε εντολή ανάθεσης, ο κόμβος i έχει c-use των μεταβλητών που υπάρχουν στο δεξιό μέλος οι οποίες ακολουθούνται από ένα ορισμό (def) της μεταβλητής στο αριστερό μέλος. Όταν υπάρχουν πολλές απλές εντολές συνεχόμενες χωρίς την παρέμβαση άλλου είδους εντολών, τότε αυτές μπορούν να συνενωθούν σε ένα μόνο κόμβο. Σε τέτοια περίπτωση ο κόμβος περιέχει c-use των μεταβλητών που υπάρχουν στο δεξιό μέλος όλων των επιμέρους εντολών και def της μεταβλητής στο αριστερό μέλος όλων των επιμέρους εντολών. Η ίδια στρατηγική ακολουθείται και για τις εντολές εισόδου ή όταν έχουμε αρχικοποίηση μεταβλητής. Σε αυτή περίπτωση ο κόμβος περιλαμβάνει μόνο def της μεταβλητής αφού το δεξιό μέλος είναι μια αριθμητική (ή άλλη) τιμή. Σημειώνεται ότι για σκοπούς συμβατότητας με τα κριτήρια και τους περιορισμούς που παρουσιάστηκαν σε προηγούμενο κεφάλαιο, ο γράφος κάθε προγράμματος ξεκινά με ένα κόμβο – source και τελειώνει με τον κόμβο sink. Τα πιο πάνω συνοψίζονται στο παράδειγμα που ακολουθεί.

```
X = 0;
Y = 5;
Z = y^2;
w = x^4;
if(…){
```

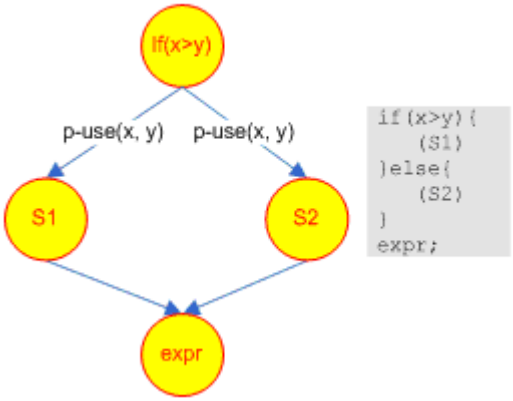
Το πιο πάνω κομμάτι κώδικα, μέχρι την εντολή if, παρουσιάζεται στον γράφο σαν ένας μοναδικό κόμβος.



Σχήμα 5.2: Αναπαράσταση του κώδικα, ορισμοί και χρήσεις

- **Εντολές Συνθήκης**

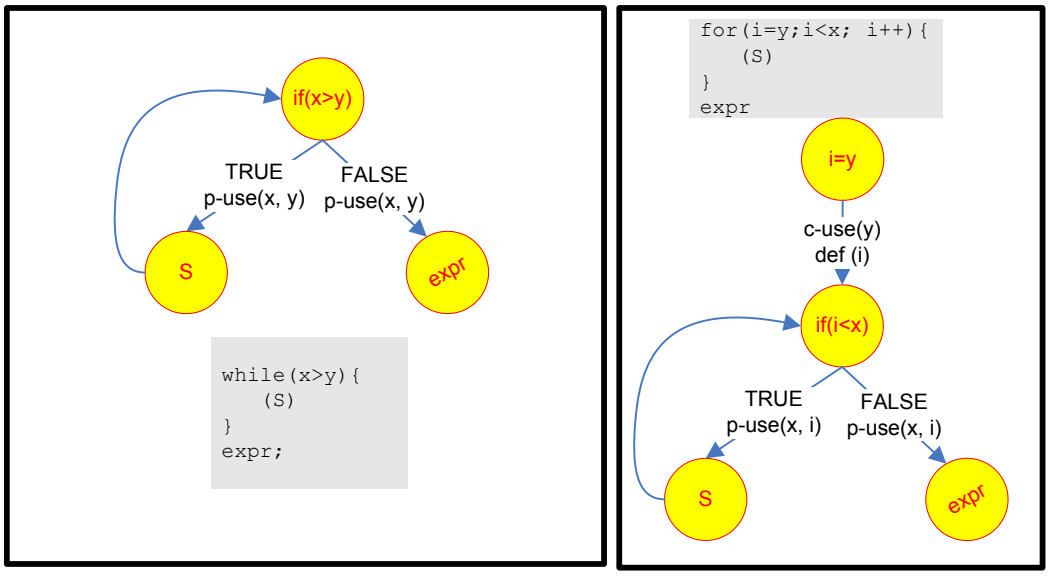
Οι εντολή συνθήκης if – then – else παρουσιάζεται σχεδόν σε κάθε γλώσσα προγραμματισμού. Ο κόμβος που ορίζει την συνθήκη περιέχει ακμές προς κάθε πιθανό μονοπάτι, το οποίο αποφασίζετε με βάση την τιμή της συνθήκης. Κάθε μια από αυτές τις εξερχόμενες ακμές περιέχει p-use των μεταβλητών που λαμβάνουν μέρος στην συνθήκη. Ακολουθεί παράδειγμα.



Σχήμα 5.3: Παράδειγμα για εντολή συνθήκης if

- **Εντολές Επανάληψης**

Οι εντολές επανάληψης που υπάρχουν σε όλες τις γλώσσες προγραμματισμού είναι οι while, for και η εντολή do – while (ή repeat – until). Σε αυτή την περίπτωση ο γράφος συμπεριφέρεται όπως στην περίπτωση εντολών συνθήκης. Δηλαδή σε κάθε εξερχόμενη ακμή του κόμβου που περιέχει την Boolean έκφραση υπάρχει p-use των μεταβλητών που συμμετέχουν στην έκφραση αυτή. Το πιο κάτω παράδειγμα δείχνει πώς μετατρέπονται οι εντολές επανάληψης με αποτέλεσμα να μοιάζουν με εντολές συνθήκης.



Σχήμα 5.4: Παραδείγματα μετατροπής while και for loops

5.3. Παρόμοιες/παλιές εργασίες – Related work

Σχετικά με το θέμα "έλεγχος λογισμικού" και "αυτόματη παραγωγή test cases" έχουν γίνει κι άλλες μελέτες. Έχω συνοψίσει κάποιες από αυτές πιο κάτω, περιγράφοντας κάπως περιληπτικά το θέμα με το οποίο ασχολείτο η κάθε μια.

Οι Abdelaziz Khamis et all στο [28] ασχοληθήκαν με την αυτόματη παραγωγή test cases για προγράμματα Pascal τα οποία ικανοποιούσαν data flow κριτήρια. Αντιθέτως με τα ήδη υπάρχοντα εργαλεία ελέγχου, το δικό τους δεν περιοριζόταν σε προγράμματα Pascal, οι γράφοι ροής των οποίων περιείχαν read statements σ' ένα μόνο κόμβο αλλά μπορούσε να χειριστεί προγράμματα στα οποία τα read statements τους μπορούσαν να εμφανίζονται σε οποιοδήποτε κόμβο. Επίσης το εργαλείο που ανέπτυξαν χειριζόταν loops και arrays, δύο χαρακτηριστικά τα οποία παραδοσιακά αποτελούν δύσκολο σημείο διαχείρισης σε συστήματα αυτόματης παραγωγής test cases. Γενικά η προσέγγιση που ανέπτυξαν υπερνικούσε τις αδυναμίες του dynamic domain reduction procedure (DDR) οι οποίες ήταν:

1) Το DDR διαχειριζόταν προγράμματα οι γράφοι των οποίων περιείχαν read statements μόνο σ' ένα κόμβο.
2) Προϋπόθετε ότι τα domains των input variables είχαν συγκεκριμένες διακριτές τιμές και
3) Το DDR δεν είχε λύσει ολοκληρωτικά το πρόβλημα με τα arrays και τα loops.

Η προσέγγιση τους είναι βασισμένη στην ιδέα διαμοίρασης των input domains του προγράμματος το οποίου είναι under testing σε υποσύνολα, τα οποία ονομάζουν

sub-domains, και ακολούθως απαιτούσαν τα rest cases να περιέχουν στοιχεία από αυτά τα domains. Η προσέγγιση αυτή παρουσιάζεται αναλυτικότερα στο [28].

Στο [29] οι Ahmed S. Chiduk et all επίσης ασχολήθηκαν με την αυτόματη παραγωγή test cases που χρησιμοποιεί γενετικούς αλγορίθμους για να παράξουν test data που ικανοποιούν data-flow coverage κριτήρια. Η τεχνική εφαρμόζει τις ιδέες των κυρίαρχων σχέσεων μεταξύ των κόμβων για να ορίσει μια νέα multi-objective fitness function για να αξιολογήσει τα παραγόμενα test data. Στο paper αυτό παρουσιάζονται τα αποτελέσματα που πάρθηκαν από ένα σύνολο εμπειρικών μελετών που διεξάχθηκαν από ένα σύνολο προγραμμάτων και αξιολογούν την αποτελεσματικότητα της τεχνικής τους με την τεχνική random. Οι μελέτες που έγιναν δείχνουν τη αποτελεσματικότητα της τεχνικής τους σε επιτυχία κάλυψης των απαιτήσεων ελέγχου, σε χρόνο και αριθμό επαναλήψεων που απαιτούταν για να ικανοποιηθούν τα data flow κριτήρια.

Ο αλγόριθμος που προτείνουν, TGGA (**T**est-data **G**eneration technique that uses a **G**enetic **A**lgorithm), παίρνει ως input το πρόγραμμα under test (P), το σύνολο των απαιτήσεων ελέγχου (TestReq) που απαιτείται από ένα data flow κριτήριο και το dominator tree (DT). Ο TGGA παίρνει τον αρχικό πληθυσμό (InitPop) παράγοντας τον τυχαία ή από τον χρήστη. Τα πεδία ορισμού και οι ακριβής προσεγγίσεις των μεταβλητών εισόδου κα οι παραμέτροι του γενετικού αλγόριθμου (π.χ. μέγιστος αριθμός generations (MaxGens), μέγεθος πληθυσμού (PopSize), πιθανότητα crossover (XP), και πιθανότητα mutation (MP)) καθορίζοντα πειραματικά από τον χρήστη. Ο TGGA επιστρέφει το σύνολο των test data (TestCover) το οποίο καλύπτει το TestReq.

Τα αποτελέσματα τους έδειξαν ότι η τεχνική του TGGA υπερτερούσε από την τεχνική random σε 8 με 9 προγράμματα που χρησιμοποίησαν στα πειράματα. Σε άλλα προγράμματα η τεχνική τους είχε την ίδια τοις εκατόν κάλυψη με την τεχνική random. Επίσης τα αποτελέσματα των πειραμάτων έδειξαν ότι η fitness function που προτείνουν είναι αρκετά ικανή να αξιολογήσει τα test data.

Στο άρθρο [30] παρουσιάζεται μια διαφορετική τεχνική κάλυψης, ονομαζόμενη model-checking-based approach, η οποία είναι για data flow testing. Στο άρθρο αυτό χαρακτηρίζουν τα data flow oriented coverage criteria σε χρονική λογική έτσι ώστε το πρόβλημα της παραγωγής test cases να περιορίζεται στο πρόβλημα εύρεσης μαρτύρων (finding witnesses) για ένα σύνολο χρονικών τύπων λογικής. Η ικανότητα των model checkers να κατασκευάζουν μάρτυρες και εξαιρέσεις παραδειγμάτων επιτρέπουν στο test generation να είναι ολοκληρωτικά

αυτοματοποιημένο. Επίσης συζητούνται ζητήματα πολυπλοκότητας χαμηλού κόστους και περιγράφονται ευρετικοί αλγόριθμοι παραγωγής test data. Η προσέγγιση τους επεξηγείται χρησιμοποιώντας CTL ως χρονική λογική και SMV ως πρότυπο ελεγκτή.

Μια ακόμη σχετική δουλειά έχει γίνει στο [31] όπου προτείνεται ένα πλαίσιο εργασίας παραγωγής test data βασισμένο σε γενετικούς αλγορίθμους. Αυτό το πλαίσιο εργασίας εμπεριέχει ένα program analyzer και ένα test case generator, οι οποίοι επικοινωνούν μεταξύ τους για να παράξουν αυτόματα test cases. Ο program analyzer εξάγει εντολές και μεταβλητές, απομονώνει μονοπάτια κώδικα και δημιουργεί control flow graphs. Ο test case generator χρησιμοποιεί δύο optimization αλγόριθμους, τον Batch-Optimistic (BO) και τον Close-Up (CU), και παράγει ένα βέλτιστο σύνολο από test cases για το κριτήριο edge/condition. Η δύναμη της προτεινόμενης προσέγγισης καθορίζεται στα διάφορα προγράμματα και τα εμπειρικά αποτελέσματα τα οποία φανερώνουν ότι η προσέγγιση αυτή είναι σημαντικά καλύτερη εν συγκρίσει με ήδη υφιστάμενες μεθόδους δυναμικής παραγωγής test data.

Κεφάλαιο 6

Αυτοματοποίηση ελέγχου

6.1. Εισαγωγικά.....93
6.2. Περιγραφή υλοποίησης κριτηρίου Ntafos και ψευδοκώδικας.....93
6.3. Περιγραφή υλοποίησης γενετικών αλγορίθμων..... 102

6.1. Εισαγωγικά

Στο κεφάλαιο αυτό ουσιαστικά θα γίνει μια περιγραφή της δουλειάς που έγινε στην εργασία αυτή. Παρακάτω περιγράφεται ο αλγόριθμος υλοποίησης του κριτηρίου Ntafos και δίδεται ένας ενδεικτικός ψευδοκώδικας υλοποίησης αυτού. Επίσης περιγράφεται ο γενετικός αλγόριθμος, το πώς είναι μοντελοποιημένα τα γονίδια, τα χρωμοσώματα και ο γονότυπος. Στη συνέχεια παρουσιάζεται περιγραφή της συνάρτησης ποιότητας (fitness function), γιατί τροποποιήθηκε από την προηγούμενη εργασία, πως αξιολογεί την κάθε λύση κ.λ.π.

6.2. Περιγραφή υλοποίησης κριτηρίου Ntafos και ψευδοκώδικας

Καταρχάς για να υλοποιηθεί το κριτήριο του Ntafos χρησιμοποιήθηκε ο αλγόριθμος αναζήτησης σε βάθος (Depth first search), τον οποίο περίγραψα στο κεφάλαιο 3. Με την χρήση αυτού του αλγορίθμου είχα την δυνατότητα επίσκεψης όλων των κόμβων ενός γράφου ροής δεδομένων.

Η γενική ιδέα ήταν ότι ξεκινούσα από ένα κόμβο, έλεγχα αν υπήρχαν definitions σε εκείνο τον κόμβο και αν υπήρχαν έπαιρνα ένα-ένα τα definitions, έλεγχα αν το search state της πρώτης ακμής που εξερχόταν από τον συγκεκριμένο κόμβο (μπορεί να υπήρχαν περισσότερες από μια ακμές που εξέρχονταν από τον συγκεκριμένο κόμβο π.χ. if statements) ήταν μηδέν, δηλαδή ότι δεν την επισκέφτηκα ξανά. Αν όντως ήταν μηδέν έλεγχα αν υπήρχαν p-uses πάνω στην συγκεκριμένη ακμή και περιείχαν την μεταβλητή του definition από το οποίο άρχισα, και αν σύμβαινε αυτό και το DR-interaction μου ήταν ένα, τότε αποθήκευα το μονοπάτι μέχρι εκείνο το σημείο και στην συνέχεια καλούσα μια αναδρομική συνάρτηση την «DFSPathDiscovery» για να μου εντοπίσει περισσότερα μονοπάτια. Αν δεν υπήρχαν p-uses πάνω στην συγκεκριμένη ακμή ή υπήρχαν και δεν περιείχαν την μεταβλητή του definition από το οποίο άρχισα τότε πάλι καλούσα την

αναδρομική συνάρτηση για να συνεχίσει να ερευνά για DR-interactions paths δια μέσου και του υπόλοιπου γράφου. Το μέγεθος των DR-interactions paths καθοριζόταν από τις αρχικές παραμέτρους που έδινε ο χρήστης.

Η αναδρομική συνάρτηση «DFSPathDiscovery» παίρνει ως παράμετρο την μεταβλητή (currentVariable) για την οποία ψάχνεται DR-interaction την συγκεκριμένη χρονική στιγμή, το τρέχων μονοπάτι (currentPath) το οποίο περιέχει τους κόμβους από τους οποίους πέρασε η συγκεκριμένη λύση μέχρι εκείνη την φάση, τον κόμβο (vertexToStart) από τον οποίο πρέπει να ξεκινήσει να ψάχνει η συνάρτηση και το DR-interaction (DrInteraction) που ψάχνεται εκείνη την δεδομένη στιγμή. Αυτές είναι οι πιο σημαντικές παραμέτροι οι οποίοι κάθε φορά που καλείται αναδρομικά η συνάρτηση αλλάζουν και οπόταν εκτελούνται διαφορετικά πράγματα.

Η συνθήκη τερματισμού της «DFSPathDiscovery» ήταν όταν έφθανε στον κόμβο με το όνομα "End". Κάθε φορά που καλείτο αυτή η συνάρτηση αποθηκευόταν στο τρέχων μονοπάτι (currentPath) ο κόμβος "vertexToStart" ούτως ώστε να μην χάνεται το τρέχον μονοπάτι από το οποίο περνά κάθε φορά η λύση.

Στην συνάρτηση αυτή γινόταν έλεγχος αν το μονοπάτι που διασχιζόταν κάθε φορά από κάποιο definition σε κάποιο use (είτε c-use είτε p-use) ήταν def-clear, δηλαδή δεν περιείχε definitions της μεταβλητής currentVariable. Αν ο κόμβος που ελεγχόταν εκείνη την στιγμή δεν περιείχε definitions της συγκεκριμένης μεταβλητής αυτό που γινόταν ήταν το εξής: έλεγχα για τα c-use του κόμβου αυτού.

Αν τα c-use περιείχαν την μεταβλητή, τότε κοίταζα και αν το συγκεκριμένο DrInteraction ήταν ένα. Αν ήταν ένα τότε αυτό σημαίνει ότι βρήκα μια αλυσίδα που ξεκινά με def της currentVariable και φθάνει σε κάποιο c-use της τρέχον μεταβλητής δια μέσου ενός path που είναι και def-clear. Τότε αποθήκευα το μονοπάτι αυτό. Στην περίπτωση τώρα που υπάρχει c-use της τρέχον μεταβλητής αλλά το DrInteraction δεν είναι ένα, ελέγχω αν στον κόμβο που βρήκα το c-use υπάρχουν definitions. Αν υπάρχουν παίρνω ένα-ένα τα definitions και ακολούθως παίρνω μια-μια τις ακμές που εξέρχονται από τον συγκεκριμένο κόμβο και κοιτάζω αν το search state της κάθε ακμής είναι μηδέν, δηλαδή ότι δεν την επισκέφτηκα ξανά (όπως προηγουμένως). Αν όντως ήταν μηδέν έλεγχα αν υπήρχαν p-uses πάνω στην συγκεκριμένη ακμή και αν τα p-uses περιείχαν την μεταβλητή, και αν την περιείχαν και το DR-interaction εκείνη την στιγμή ήταν ένα, τότε αποθήκευα το μονοπάτι μέχρι εκείνο το σημείο ειδάλλως καλούσα ξανά με αναδρομή την συνάρτηση «DFSPathDiscovery» για να μου εντοπίσει περισσότερα μονοπάτια αλλά τώρα με DrInteraction πλην 1, επειδή ήδη έχω βρει ένα def που οδηγείται σε c-use. Αν τα p-uses τώρα δεν περιείχαν την μεταβλητή τότε πάλι καλούσα την αναδρομική συνάρτηση για να συνεχίσει να ερευνά για DR-interactions paths δια μέσου και του

υπόλοιπου γράφου. Στην συνέχεια επαναλάμβανα την ίδια διαδικασία για τις ακμές του κόμβου vertexToStart για να ελέγχω και τις ακμές του συγκεκριμένου κόμβου. Ξέχασα να αναφέρω, αν και είναι εύλογο, ότι κάθε φορά που καλείτο η αναδρομή καλείτο με τον επόμενο κόμβο πάνω στον οποίο κατευθυνόταν κάθε φορά η συγκεκριμένη ακμή. Όταν η αναδρομή έφτανε σε αδιέξοδο ή συνθήκη τερματισμού γινόταν το λεγόμενο backtracking-οπισθοδρόμηση.

Ας περιγράψω τώρα τι γινόταν στην περίπτωση που υπήρχαν definitions σε εκείνο τον κόμβο που έλεγχα. Με λίγα λόγια το μονοπάτι που ακολουθούταν δεν είναι def-clear. Σε αυτή την περίπτωση αν το DrInteraction ήταν ένα τότε έλεγχα αν ο κόμβος περιείχε c-use. Αν περιείχε τότε το μονοπάτι που ακολουθούταν ήταν επιτρεπτό αφού παρόλο που υπάρχει definition στον ίδιο κόμβο η μεταβλητή δεν προλαβαίνει να αλλάξει τιμή αφού πρώτα θα εκτελεστεί το δεξιό κομμάτι της εξίσωσης που περιέχει το c-use και στη συνέχεια θα γίνει το definition και θα αλλάξει τιμή η μεταβλητή. Οπόταν αποθήκευα το μονοπάτι.

Αν τώρα δεν ήταν ένα το DrInteraction πάλι έλεγχα αν ο κόμβος περιείχε c-use. Αν περιείχε τότε η αλυσίδα def μέχρι c-use ήταν επιτρεπτή και στη συνέχεια έλεγχα αν υπήρχαν definitions σε εκείνο τον κόμβο και αν υπήρχαν έπαιρνα ένα-ένα τα definitions, έλεγχα αν το search state της κάθε ακμής που εξερχόταν από τον συγκεκριμένο κόμβο ήταν μηδέν, δηλαδή ότι δεν την επισκέφτηκα ξανά. Αν όντως ήταν μηδέν έλεγχα αν υπήρχαν p-uses πάνω στην συγκεκριμένη ακμή και περιείχαν την μεταβλητή του definition από το οποίο άρχισα, και αν σύμβαινε αυτό και το DR-interaction μου ήταν ένα, τότε αποθήκευα το μονοπάτι μέχρι εκείνο το σημείο. Αν δεν υπήρχαν p-uses πάνω στην συγκεκριμένη ακμή ή υπήρχαν και δεν περιείχαν την μεταβλητή του definition από το οποίο άρχισα τότε πάλι καλούσα την αναδρομική συνάρτηση για να συνεχίσει να ερευνά για DrInteraction πλην 1 paths δια μέσου και του υπόλοιπου γράφου.

Αυτή είναι κάπως περιληπτικά η διαδικασία που χρησιμοποιείται για την υλοποίηση του κριτηρίου του Ntafos. Να αναφέρω ότι για μεγαλύτερου αριθμού DR-interactions υπάγονται και τα μικρότερου βαθμού DR-interactions. Π.χ. για το interaction = 3, τότε θα έχω μονοπάτια βαθμού 3, 2 και 1. Αν φυσικά υπάρχουν στον γράφο τέτοιου βαθμού interactions. Το σίγουρο είναι πως υπάρχουν interactions τουλάχιστον βαθμού 1 για τους περισσότερους γράφους. Το να υπάγονται και τα μικρότερου βαθμού DR-interactions επιτυγχάνεται με την χρήση ενός for loop, έξω από την αναδρομική συνάρτηση, το οποίο loop περιέχει αυτά που περίγραψα στην δεύτερη παράγραφο.

Πιο κάτω παρουσιάζω με ψευδοκώδικα τον αλγόριθμο που μόλις περίγραψα:

```
k = getKDRInteraction();
WHILE k>0
        Απαρίθμησε τους κόμβους του γράφου ροής δεδομένων
        WHILE υπάρχουν κι άλλοι κόμβοι
                Πάρε τον currentVertex
                Πάρε τα definitions του currentVertex
                WHILE υπάρχουν definitions του currentVertex
                        Πάρε το definition στο currentVariable
                        Όρισε ότι όλες οι ακμές δεν είναι visited ακόμη
                        Όρισε ένα currentPath
                        Πρόσθεσε στο currentPath τον αριθμό του κόμβου που
                        βρίσκεσαι
                        Απαρίθμησε τις ακμές του κόμβου που βρίσκεσαι
                        WHILE υπάρχουν κι άλλες ακμές
                                Πάρε την ακμή στο outEdge
                                Πάρε τον κόμβο στον οποίο φθάνει η ακμή στο
                                followingVertex
                                Όρισε ότι όλες οι ακμές μετά την ακμή outEdge  δεν
                                είναι visited ακόμη
                                IF η outEdge δεν είναι visited THEN
                                        Θέσε ότι η outEdge έγινε visited
                                        Πάρε τα p-use της outEdge  στο
                                        pUseVariablesOfEdge
                                        IF το pUseVariablesOfEdge δεν περιέχει την
                                        currentVariable THEN
                                                DFSPathDiscovery(currentVariable,
                                                currentPath, vertexToStart, k);
                                                Αφαίρεσε από το currentPath τον
                                                τελευταίο κόμβο
                                        ELSE
                                                IF k=1 THEN
                                                        Πρόσθεσε στο currentPath τον
                                                        αριθμό του κόμβου που βρίσκεσαι
                                                        Πρόσθεσε το currentPath στο
                                                        pathsInfo
```

```
                                        Αφαίρεσε από το currentPath τον
                                        αριθμό του κόμβου που βρίσκεσαι
                                        DFSPathDiscovery(currentVariabl
                                        e, currentPath, vertexToStart, k);
                                        Αφαίρεσε από το currentPath τον
                                        τελευταίο κόμβο
                                    ELSE
                                        DFSPathDiscovery(currentVariabl
                                        e, currentPath, vertexToStart, k);
                                        Αφαίρεσε από το currentPath τον
                                        τελευταίο κόμβο
                                    ENDIF
                                ENDIF
                            ENDIF
                            Πάρε την επόμενη ακμή
                        ENDWHILE
                        Πάρε το επόμενο definition του currentVertex
                ENDWHILE
                Πάρε τον επόμενο κόμβο
        ENDWHILE
        k = k - 1
ENDWHILE


function DFSPathDiscovery(currentVariable, currentPath, vertexToStart,
DrInteraction)
{
        Πάρε τα definitions του vertexToStart στο DefsVariablesOfNode
        Πρόσθεσε στο currentPath τον αριθμό του κόμβου vertexToStart
        IF το όνομα του κόμβου είναι "End" THEN
                return;
        ENDIF

        IF το DefsVariablesOfNode δεν περιέχει την currentVariable THEN
                Πάρε τα c-use στο του vertexToStart στο cUseVariablesOfNode
                IF το cUseVariablesOfNode περιέχει την currentVariable THEN
                        IF DrInteraction = 1 THEN
```

```
            Πρόσθεσε το currentPath στο pathsInfo
ELSE
            Πάρε τα definitions του vertexToStart στο
            DefsVariablesOfNode2
            FOR όσα είναι το size του DefsVariablesOfNode2
                  Πάρε το definition στο currentVariable2
                  Απαρίθμησε τις ακμές του κόμβου που βρίσκεσαι
                  WHILE υπάρχουν κι άλλες ακμές
                        Πάρε την ακμή στο outEdge
                        Πάρε τον κόμβο στον οποίο φθάνει η
                        ακμή στο followingVertex
                        Όρισε ότι όλες οι ακμές μετά την ακμή
                        outEdge  δεν είναι visited ακόμη

                        IF η outEdge δεν είναι visited THEN
                              Θέσε ότι η outEdge έγινε visited
                              Πάρε τα p-use της outEdge  στο
                              pUseVariablesOfEdge

                        IF το pUseVariablesOfEdge δεν περιέχει
                        την currentVariable THEN
                              DFSPathDiscovery(currentVariabl
                              e, currentPath, vertexToStart, --
                              DrInteraction);
                              Αφαίρεσε από το currentPath τον
                              τελευταίο κόμβο
                              DrInteraction=DrInteraction+1
                        ELSE
                              IF (DrInteraction-1)=1 THEN
                                    Πρόσθεσε στο currentPath
                                    τον αριθμό του κόμβου
                                    που βρίσκεσαι
                                    Πρόσθεσε το currentPath
                                    στο pathsInfo
                              ELSE
```

```
                                                    DFSPathDiscovery(current
                                                    Variable, currentPath,
                                                    vertexToStart,
                                                    --DrInteraction);
                                                    Αφαίρεσε από το
                                                    currentPath τον τελευταίο
                                                    κόμβο
                                                ENDIF
                                            ENDIF
                                        ENDIF
                                        Πάρε την επόμενη ακμή
                                        ENDWHILE
                                ENDFOR
                        ENDIF
                ENDIF
        ELSE
        Απαρίθμησε τις ακμές του κόμβου που βρίσκεσαι
        WHILE υπάρχουν κι άλλες ακμές
                Πάρε την ακμή στο outEdge
                Πάρε τον κόμβο στον οποίο φθάνει η ακμή στο followingVertex

                IF η outEdge δεν είναι visited THEN
                        Θέσε ότι η outEdge έγινε visited
                        Πάρε τα p-use της outEdge  στο pUseVariablesOfEdge
                        IF το pUseVariablesOfEdge δεν περιέχει την currentVariable
                        THEN
                                DFSPathDiscovery(currentVariable, currentPath,
                                vertexToStart, DrInteraction);
                                Αφαίρεσε από το currentPath τον τελευταίο κόμβο
                        ELSE
                                IF DrInteraction =1 THEN
                                        Πρόσθεσε στο currentPath τον αριθμό του
                                        κόμβου που βρίσκεσαι
                                        Πρόσθεσε το currentPath στο pathsInfo
                                        Αφαίρεσε από το currentPath τον αριθμό του
                                        κόμβου που βρίσκεσαι
```

```
                                    DFSPathDiscovery(currentVariable, currentPath,
                                    vertexToStart, DrInteraction);
                                    Αφαίρεσε από το currentPath τον τελευταίο
                                    κόμβο
                            ELSE
                                    DFSPathDiscovery(currentVariable, currentPath,
                                    vertexToStart, DrInteraction);
                                    Αφαίρεσε από το currentPath τον τελευταίο
                                    κόμβο
                            ENDIF
                    ENDIF
            ENDIF
            Πάρε την επόμενη ακμή
    ENDWHILE
ELSE
    Πάρε τα c-use στο του vertexToStart στο cUseVariablesOfNode
    IF DrInteraction =1 THEN
            IF το cUseVariablesOfNode περιέχει την currentVariable THEN
                    Πρόσθεσε το currentPath στο pathsInfo
            ENDIF
    ELSE
            IF το cUseVariablesOfNode περιέχει την currentVariable THEN
                    Πάρε τα definitions του vertexToStart στο
                    DefsVariablesOfNode2
                    FOR όσα είναι το size του DefsVariablesOfNode2
                            Πάρε το definition στο currentVariable2
                            Απαρίθμησε τις ακμές του κόμβου που βρίσκεσαι
                            WHILE υπάρχουν κι άλλες ακμές
                                    Πάρε την ακμή στο outEdge
                                    Πάρε τον κόμβο στον οποίο φθάνει η ακμή στο
                                    followingVertex
                                    Όρισε ότι όλες οι ακμές μετά την ακμή outEdge
                                    δεν είναι visited ακόμη

                                    IF η outEdge δεν είναι visited THEN
                                            Θέσε ότι η outEdge έγινε visited
```

```
                                        Πάρε τα p-use της outEdge  στο
                                        pUseVariablesOfEdge
                                        IF το pUseVariablesOfEdge δεν περιέχει
                                        την currentVariable THEN
                                                DFSPathDiscovery(currentVariabl
                                                e, currentPath, vertexToStart, --
                                                DrInteraction);
                                                Αφαίρεσε από το currentPath τον
                                                τελευταίο κόμβο
                                                DrInteraction=DrInteraction+1
                                        ELSE
                                                IF DrInteraction=1 THEN
                                                        Πρόσθεσε στο currentPath
                                                        τον αριθμό του κόμβου
                                                        που βρίσκεσαι
                                                        Πρόσθεσε το currentPath
                                                        στο pathsInfo
                                                ELSE
                                                        DFSPathDiscovery(current
                                                        Variable, currentPath,
                                                        vertexToStart,
                                                        DrInteraction);
                                                        Αφαίρεσε από το
                                                        currentPath τον τελευταίο
                                                        κόμβο
                                                ENDIF
                                        ENDIF
                                ENDIF
                                Πάρε την επόμενη ακμή
                        ENDWHILE
                ENDFOR
        ENDIF
ENDIF

}
```

6.3. Περιγραφή υλοποίησης γενετικών αλγορίθμων

Στην εργασία αυτή με τη βοήθεια των γενετικών αλγορίθμων επιλύθηκε το πρόβλημα παραγωγής τιμών εισόδου για κάποιο υπό έλεγχο πρόγραμμα. Ο κάθε γενετικός αλγόριθμος περιλαμβάνει και ορίζει το γονίδιο, το χρωματόσωμα, το γονότυπο και την συνάρτηση ποιότητας (fitness function). Ακολουθεί μια γενική περιγραφή της δομής του κάθε ενός από τα τέσσερα στοιχεία του γενετικού αλγόριθμού όπως χρησιμοποιήθηκε στην εργασία αυτή. Ο τρόπος που είναι υλοποιημένα όλα τα στοιχεία του γενετικού αλγορίθμου, επιτρέπει την δημιουργία συνάρτησης ποιότητας και επιλογής χωρίς να χρειάζεται αλλαγή στον κύριο κώδικα του ίδιου του γενετικού αλγορίθμου.

6.3.1. Γονίδιο (gene)

Τα γονίδια αντιπροσωπεύουν το μικρότερο συστατικό μιας πιθανής λύσης. Ανάλογα με την περίπτωση, τα γονίδια μπορεί να αντιπροσωπεύουν πραγματικούς αριθμούς, τιμές αλήθειας, δυαδικές αναπαραστάσεις κτλ. Η δομή του στην περίπτωση μας είναι γενική και μπορεί να πάρει οποιαδήποτε μορφή και να χρησιμοποιηθεί χωρίς σημαντικές αλλαγές στον κώδικα αφού το κάθε γονίδιο θα κληρονομεί από τη αρχική κλάση την Gene η οποία βρίσκεται στο πακέτο γενετικών αλγορίθμων «JGAP».

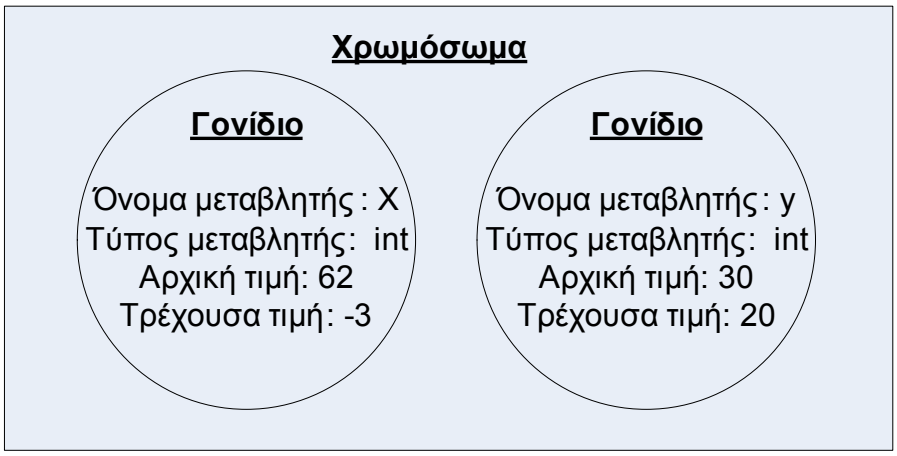
Για τη συγκεκριμένη μελέτη, το γονίδιο σχεδιάστηκε ώστε να αντιπροσωπεύει μια μεταβλητή εισόδου. Εκτός από το όνομα και την τιμή της μεταβλητής, το κάθε γονίδιο περιλαμβάνει την μέγιστη και την ελάχιστη τιμή που μπορεί να πάρει η μεταβλητή (αν αυτό είναι εφαρμόσιμο – π.χ. σε ακεραίους). Στην περίπτωση μας όμως αγνοούμε τα όρια αυτά για τον λόγο ότι στο mutation δημιουργώ δυναμικά την μετάλλαξη ανάλογα με το evolution που βρίσκεται εκείνη την ώρα ο αλγόριθμος δίχως να ελέγχω τα όρια. Αυτό είναι πιο αποδοτικό γιατί στο mutation αυξάνονται σταδιακά οι τιμές και είναι πιο πιθανό να πέσουμε στις τιμές που επιθυμούμε.

Εικόνα 7.1: Μοντελοποίηση δομής γονιδίου

Για κάθε πρόγραμμα, η δομή δεδομένων που αντιπροσωπεύει ένα γονίδιο δημιουργείται δυναμικά, αφού κατά την ανάλυση του προγράμματος εντοπίζονται οι παράμετροι εισόδου. Το κάθε γονίδιο κληρονομεί από την αρχική κλάση (Gene) όλες τις μεθόδους και ιδιότητες ώστε να είναι εφικτό να εφαρμοστούν σε αυτό πράξεις όπως για παράδειγμα το crossover και mutation.

6.3.2. Χρωμόσωμα (Chromosome)

Ένα χρωμόσωμα δεν είναι τίποτε άλλο από ένα σύνολο από Ν γονίδια, με το Ν να είναι ο συνολικός αριθμός μεταβλητών εισόδου που έχει το εξεταζόμενο πρόγραμμα. Επομένως τα χρωμοσώματα αποτελούνται από μια καθορισμένου μήκους συλλογή των γονιδίων και αντιπροσωπεύουν μια πιθανή λύση. Ο αριθμός των γονιδίων που έχει κάθε χρωμόσωμα είναι ο ίδιος για όλο τον πληθυσμό. Με βάση την υλοποίηση του γονιδίου, το i-οστό γονίδιο του κάθε χρωματοσώματος, πρέπει να έχει ίδιο όνομα μεταβλητής και όρια μέγιστης και ελάχιστης τιμής με το γονίδιο στην αντίστοιχη θέση σε κάποιο άλλο χρωμόσωμα, ενώ η τιμή της μεταβλητής μπορεί να είναι διαφορετική.



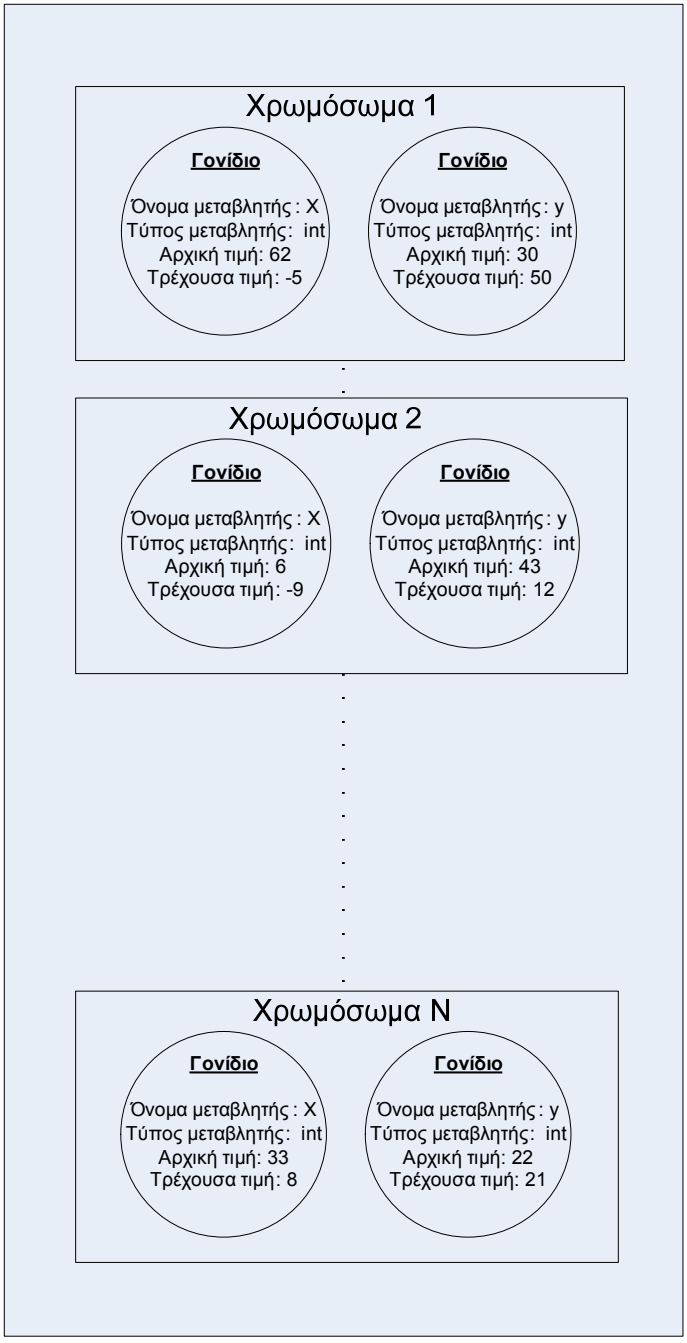
Σχήμα 7.2: Χρωμόσωμα για ένα πρόγραμμα με δύο μεταβλητές εισόδου, την x και y

Είναι εύλογο λοιπόν ότι η τελική λύση θα αποτελείται από πολλά χρωμοσώματα. Κάθε χρωμόσωμα είναι ικανό να καλύψει μόνο μέρος των μονοπατιών. Τα χρωμοσώματα τα οποία μπορούν να καλύψουν ένα συγκεκριμένο αριθμό από τα ζητούμενα, με βάση το κριτήριο, μονοπάτια αποθηκεύονται και «μαρκάρονται» ως μέρος τη τελικής λύσης.

6.3.3. Γονότυπος (Genotype)

Ο Γονότυπος ή αλλιώς πληθυσμός είναι καθορισμένου μήκους αριθμός χρωμοσωμάτων. Οι γονότυποι, και τα χρωματοσώματα τα οποία περιλαμβάνουν,

εξελίσσονται σε κάθε φάση εκτέλεσης με σκοπό να βρεθούν τα καλύτερα χρωματοσώματα τα οποία θα περάσουν στην επόμενη γενιά. Η τελική λύση στην δική μας περίπτωση μπορεί να χαρακτηριστεί ως ο «ιδανικός γονότυπος» αφού θα περιλαμβάνει τα χρωματοσώματα τα οποία είναι ικανά να μας δώσουν την λύση. Ο γονότυπος της λύσης είναι ιδανικός αλλά όχι ο βέλτιστος με την έννοια ότι μπορεί να μείνουν εκτός της λύσης πιο υγιή χρωματοσώματα και να συμπεριληφθούν άλλα που είναι λιγότερο υγιή αφού τα τελευταία μπορεί να προσφέρουν κάλυψη μονοπατιών που άλλα χρωματοσώματα αδυνατούν να καλύψουν.



Σχήμα 7.3: Δομή Γονότυπου για πληθυσμό N χρωμοσωμάτων

6.3.4. Συνάρτηση Ποιότητας (Fitness function)

Σκοπός των συναρτήσεων ποιότητας είναι να καθορίσουν πόσο καλή είναι μια συγκεκριμένη λύση συγκριτικά με άλλες. Στη συνάρτηση δίνεται ένα χρωμόσωμα ως παράμετρος με σκοπό να το αξιολογήσει και να επιστρέψει έναν θετικό ακέραιο αριθμό που να απεικονίζει την αξία ή υγεία του. Όσο ψηλότερη η αξία, τόσο ικανότερο θεωρείται το χρωμόσωμα. Δύο χρωματοσώματα με τα ίδια σύνολα γονιδίων πρέπει πάντα να έχουν την ίδια αξία ικανότητας.

Η συνάρτηση ποιότητας που δημιουργήθηκε σ' αυτή την εργασία διαφέρει από την συνάρτηση ποιότητας που δημιουργήθηκε στην εργασία στο [5]. Η συνάρτηση ποιότητας στην εργασία αυτή έχει τον εξής στόχο: Μονοπάτια το οποία επιφανειακά φαίνεται πως δεν καλυφτήκαν, ενώ στην ουσία έχουν καλυφθεί από κάποιο συγκεκριμένο χρωμόσωμα, τότε να τα «μαρκάρει» ως TRUE δηλαδή ότι καλύφθηκαν. Αυτό επιτυγχάνεται με την εξέταση των κόμβων που καλύπτει μια συγκεκριμένη λύση. Αν για παράδειγμα η λύση καλύπτει όλους τους κόμβους ενός συγκεκριμένου μονοπατιού αλλά όχι με την σειρά που εμφανίζονται στο συγκεκριμένο μονοπάτι αυτό δεν σημαίνει κατά ανάγκη ότι το μονοπάτι δεν καλύφθηκε. Μπορεί ενδιάμεσα των κόμβων του μονοπατιού να εμφανίζονται κι άλλοι κόμβοι, λόγω π.χ. κάποιου loop στο πρόγραμμα, αλλά αυτοί οι ενδιάμεσοι κόμβοι να είναι def-clear και έτσι να μην μας επηρεάζουν και το μονοπάτι να είναι TRUE δηλαδή να καλύφθηκε. Αυτή είναι η κύρια λειτουργία που γίνεται στην συνάρτηση ποιότητας.

Αυτό που γίνεται πιο συγκεκριμένα είναι ότι όταν ο γενετικός αλγόριθμος ξεκινά την λειτουργία του κάνει focus, «εστιάζει» πάνω σ' ένα μονοπάτι. Για το συγκεκριμένο μονοπάτι εκτελεί ένα αριθμό evolutions (συνήθως καθορίζεται από τον χρήστη στο ξεκίνημα του γενετικού) και προσπαθεί μέσα σ' αυτό το διάστημα να καλύψει το συγκεκριμένο μονοπάτι. Στην προσπάθεια του αυτή να καλύψει το μονοπάτι μπορεί τα χρωμοσώματα (οι συγκεκριμένες λύσεις δηλαδή) που εμφανίζονται να πετυχαίνουν κάλυψη κι άλλων μονοπατιών. Αν τύχει αυτό τότε τα συγκεκριμένα χρωμοσώματα αποθηκεύονται ως υπεύθυνα για την κάλυψη των συγκεκριμένων μονοπατιών, παρόλο που στόχος είναι να καλυφθεί το μονοπάτι που έγινε focus. Αυτό συμβαίνει λόγω του ότι τα συγκεκριμένα χρωμοσώματα μπορεί να τύχει να μην ξαναεμφανιστούν και έτσι να χάσουμε την κάλυψη κάποιου μονοπατιού που στην ουσία την καλύψαμε. Έτσι με αυτό τον τρόπο έχουμε πολυδιάστατη κάλυψη ενώ ταυτόχρονα είμαστε εστιασμένοι σε κάποιο μονοπάτι. Αν τώρα το μονοπάτι το οποίο έγινε focus καλύφθηκε με πιο λίγα evolutions από αυτά που του καθοριστήκαν αρχικά τότε σταματούν τα evolutions για το συγκεκριμένο μονοπάτι και το ψάξιμο επικεντρώνεται πάνω σε άλλο μονοπάτι, γίνεται η «εστίαση» δηλαδή σε άλλο μονοπάτι. Στην περίπτωση τώρα που και το επόμενο μονοπάτι έχει καλυφθεί από κάποιο χρωμόσωμα, λόγω προηγούμενων evolutions, τότε

δεν εκτελούνται evolutions για το συγκεκριμένο μονοπάτι αφού το μονοπάτι ήδη καλύφθηκε. Στο τέλος αυτό που επιστρέφει η συνάρτηση ποιότητας μετά από κάθε evolution είναι ο αριθμός των κόμβων του μονοπατιού που έγινε «focus» οι οποίοι καλύφθηκαν από την συγκεκριμένη λύση που δίνει το συγκεκριμένο χρωμόσωμα ως προς τον αριθμό όλων των κόμβων που περιέχει το μονοπάτι. Το fitness με λίγα λόγια επιστρέφει ένα αριθμό από 0-1, όπου 1 είναι το καλύτερο fitness και σημαίνει το μονοπάτι έχει καλυφθεί.

Με κάπως μαθηματικό απλό τρόπο το fitness function θα μπορούσε να εκφραστεί ως εξής:

F = Κόμβοι που καλύφθηκαν σε κάποιο μονοπάτι / Σύνολο όλων των κόμβων του μονοπατιού

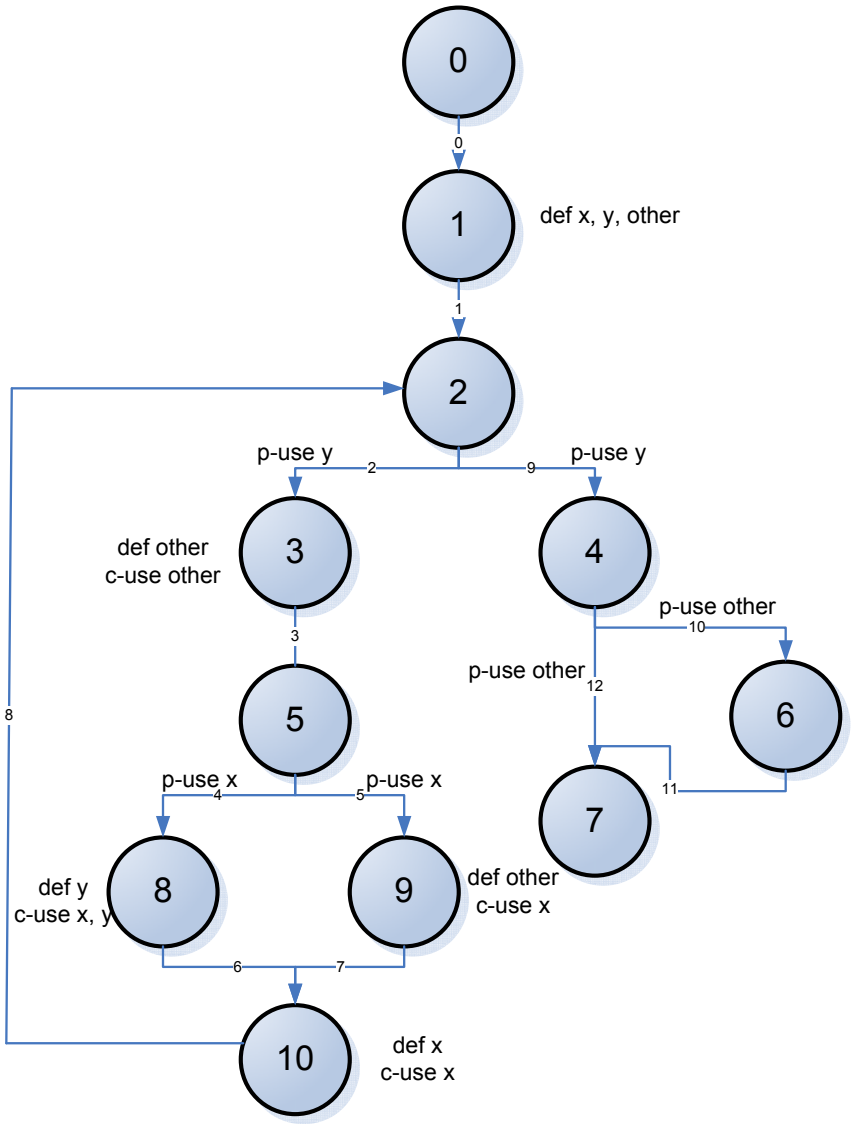
Όπου:

Κόμβοι που καλύφθηκαν σε κάποιο μονοπάτι: ο αριθμός των κόμβων που μετρήθηκαν ότι καλύφθηκαν μέσα από την συγκεκριμένη λύση όπως περιγράφτηκε πιο πάνω.

Σύνολο κόμβων του μονοπατιού: όλοι οι κόμβοι του «εστιασμένου» μονοπατιού.

Το συγκεκριμένο fitness function δεν έχει την αδυναμία να κολλήσει σε κάποιο μονοπάτι το οποίο για παράδειγμα είναι dead code και δεν πρόκειται να καλυφθεί ποτέ. Αυτό συμβαίνει επειδή: παρόλο που κάνει focus σε ένα μονοπάτι και τρέχει κάποια evolutions μέχρι να το καλύψει, δεν μένει εκεί επ’ άπειρο μέχρι να το καλύψει αλλά προχωρά σε επόμενα μονοπάτια όταν τα συγκεκριμένα evolutions τελειώσουν. Έτσι μ’ αυτό τον τρόπο του δίνει αρκετή ευκαιρία για να το καλύψει αλλά ταυτόχρονα αποφεύγει και τον άπειρο βρόγχο που θα προκαλούσε ένα μονοπάτι dead code.

Παρακάτω παρουσιάζω σχηματικά πως θα μπορούσε να αναπαρασταθεί το συγκεκριμένο fitness function. Στο σχήμα 7.4 φαίνεται ένας ενδεικτικός γράφος με τα def και uses του, ο οποίος περιέχει και loop.



Σχήμα 7.4: Γράφος ροής δεδομένων με τα αντίστοιχα defs και use του

Έστω τώρα ότι έχω το συγκεκριμένο 2-dr interaction def1(x), use8(x), def8(y), use8(y) το οποίο μου δίνει το μονοπάτι 1, 2, 3, 5, 8, 10, 2, 3, 5, 8. Το interaction και το μονοπάτι που παίρνω είναι τυχαίο και για χάρη παραδείγματος. Θα μπορούσαν κάλλιστα να παραχθούν και άλλα dr interaction μεγαλύτερου και μικρότερου βαθμού, τα οποία θα ικανοποιούσαν το κριτήριο του Ntafos.

Οπόταν τώρα έχω το εξής μονοπάτι:

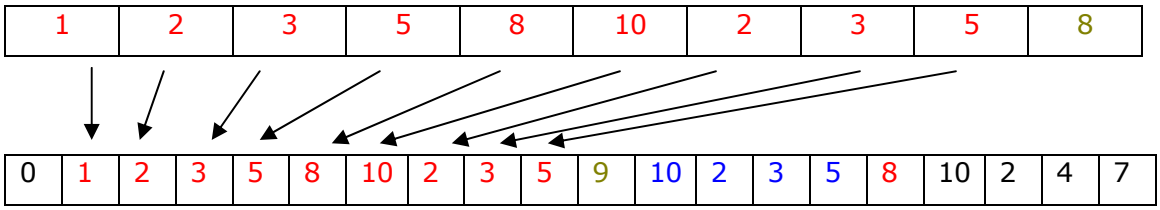
1	2	3	5	8	10	2	3	5	8

Αν τώρα ένα συγκεκριμένο χρωμόσωμα κάλυπτε την παρακάτω ακολουθία κόμβων, τότε «επιφανειακά» θα λέγαμε ότι το συγκεκριμένο μονοπάτι δεν καλύφτηκε ενώ στην ουσία έχει καλυφθεί.

Ακολουθία κόμβων που κάλυψε κάποιο χρωμόσωμα:

0	1	2	3	5	9	10	2	3	5	9	10	2	3	5	8	10	2	4	7

Παρατηρούμε:



1, 2, 3, 5, 8, 10, 2, 3, 5 : κόμβοι που καλύφθηκαν με την σειρά σύμφωνα με το συγκεκριμένο μονοπάτι.
9, 10, 2, 3, 5 : κόμβοι που είναι def clear ως προς την μεταβλητή που αναζητούμε.

Από τους 10 κόμβους που έχει το μονοπάτι καλύφθηκαν οι 9, άρα μέχρι τώρα πετύχαμε κάλυψη 9/10. Ο επόμενος κόμβος της ακολουθίας που κάλυψε το χρωμόσωμα όμως βλέπουμε ότι δεν είναι ο ίδιος με τον κόμβο που ακολουθεί στο μονοπάτι. Παρόλα αυτά δεν σημαίνει ότι το μονοπάτι δεν έχει καλυφθεί.
Ξέρουμε ότι το μονοπάτι είναι για το εξής interaction: def1(x), use8(x), def8(y), use8(y). Οπόταν μέχρι τώρα μόνο τους κόμβους των def1(x), use8(x) και def8(y) έχουμε καλύψει. Αυτό που κάνουμε τώρα είναι να ελέγξουμε τους παρακάτω κόμβους αν είναι def clear ως προς την μεταβλητή που αναζητούμε. Τώρα η μεταβλητή που αναζητούμε είναι η y, επομένως ελέγχουμε αν ο κόμβος 9 έχει def9(y), βλέπουμε πως δεν έχει και προχωρούμε στους επόμενους κόμβους. Οι κόμβοι 10, 2, 3, 5 πάλι είναι def-clear ως προς y οπότε τους διαπερνώ και φθάνω στον 8 που είναι αυτός που αναζητώ, οπόταν πέτυχα 100% κάλυψη αφού βρήκα 10 από τους 10 κόμβους. Έτσι η fitness function επιστρέφει ένα (1) που σημαίνει ότι το μονοπάτι καλύφθηκε.

Κεφάλαιο 7

Εγχειρίδιο χρήσης (Manual)

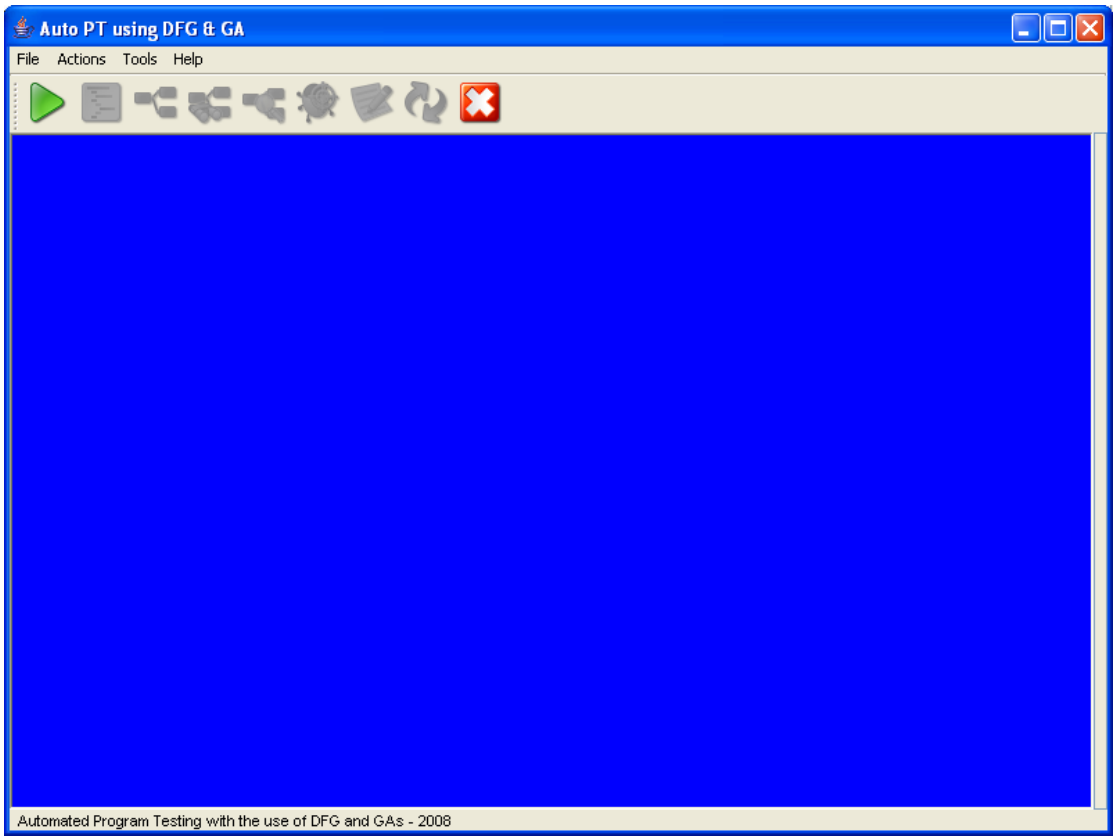
7.1. Εισαγωγή 109
7.2. Περιγραφή οθόνων 109

7.1. Εισαγωγή

Στο κεφάλαιο αυτό θα γίνει μια λεπτομερής περιγραφή του εργαλείου «αυτόματου ελέγχου» και πως μπορεί ένας απλός χρήστης να το χρησιμοποιεί. Το εργαλείο είναι ιδιαίτερα εύχρηστο και έχει δημιουργηθεί με τέτοιο τρόπο ώστε να μην απαιτούνται ιδιαίτερες γνώσεις πληροφορικής από τον χρήστη για να το χρησιμοποιεί. Γενικά το όλο interface που χρησιμοποιείται σ' αυτή την εργασία είναι το interface που χρησιμοποιήθηκε και σε προηγούμενες εργασίες τροποποιημένο με ελάχιστες αλλαγές που εξυπηρετούν την υφιστάμενη εργασία.

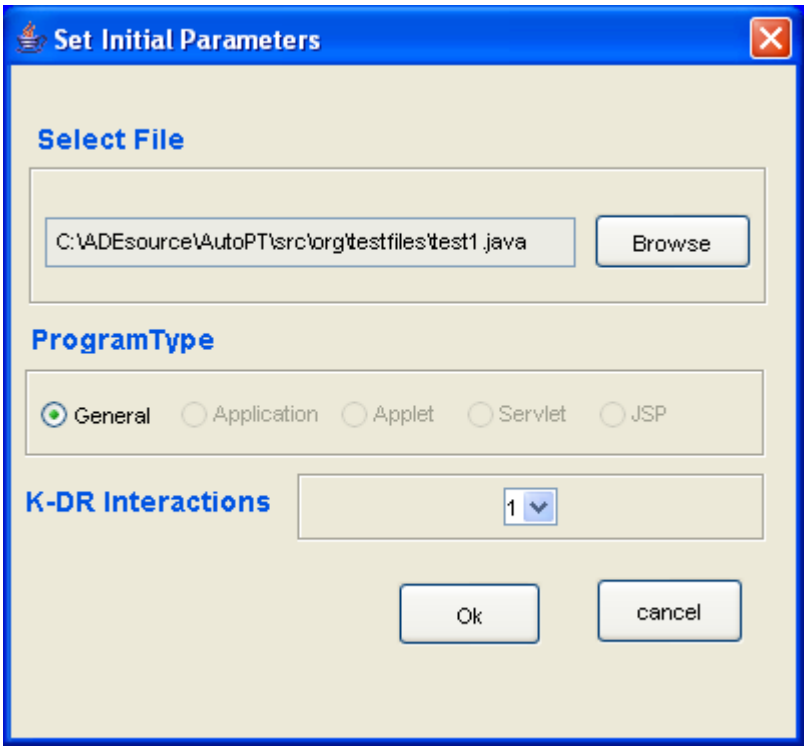
7.2. Περιγραφή οθόνων

Πιο κάτω παρουσιάζω τι πρέπει κάνει ο χρήστης κατά τη διάρκεια χρησιμοποίησης του εργαλείου αυτόματης παραγωγής δεδομένων ελέγχου. Με την εκκίνηση του προγράμματος παρουσιάζεται η πιο κάτω διεπιφάνεια (interface) προς τον χρήστη (σχήμα 6.1).



Σχήμα 6.1: Αρχική διεπιφάνεια που παρουσιάζεται προς το χρήστη (κύρια οθόνη)

Ο χρήστης πρέπει υποχρεωτικά να επιλέξει το πρόγραμμα το οποίο θέλει να ελέγξει χρησιμοποιώντας το κουμπί  . Αν δεν το κάνει δεν μπορεί να προχωρήσει σε επόμενα βήματα. Ο χρήστης έχει την δυνατότητα να πλοηγηθεί (browse) στα αρχεία του υπολογιστή του και να επιλέξει το αρχείο που θέλει (σχήμα 6.2). Στο στάδιο αυτό ο χρήστης πρέπει να επιλέξει και το k, η οποία είναι η παράμετρος που καθορίζει το k-dr interaction για την παραγωγή των μονοπατιών. Όταν το αρχείο φορτωθεί τότε ο χρήστης έχει την δυνατότητα να δει τον πηγαίο κώδικα, τον γράφο ροής ελέγχου, τον γράφο ροής δεδομένων και, το σημαντικότερο, να δει στοιχεία σχετικά με την data flow ανάλυση του προγράμματος.



Σχήμα 6.2: Ανάθεση αρχικών παραμέτρων

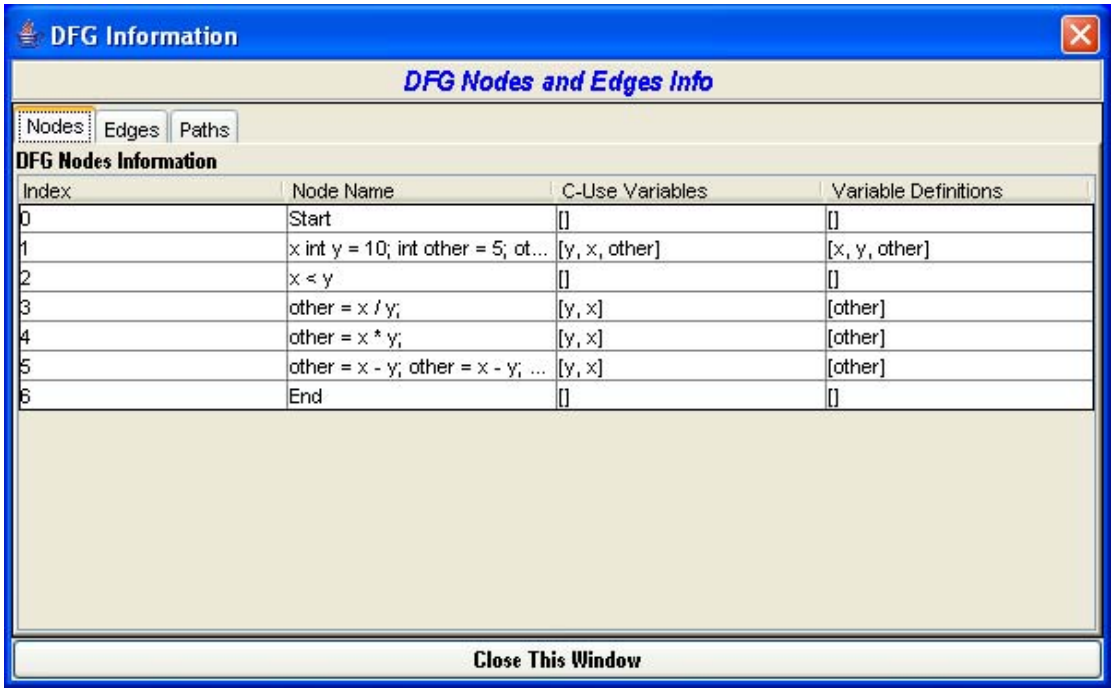
Το γραφικό περιβάλλον που δημιουργήθηκε για τον χρήστη είναι σχετικά απλό και καθοδηγεί τον χρήστη, αποτρέποντας τον από το να κάνει μια ενέργεια προτού όλα τα προηγούμενα απαιτούμενα βήματα έχουν ολοκληρωθεί. Οι επιλογές που δεν είναι δυνατόν να εκτελεστούν είναι απενεργοποιημένες τόσο στην μπάρα επιλογής ενέργειας όσο και στα μενού του προγράμματος. Μια απεικόνιση της μπάρας ενεργειών ακολουθεί.

Σχήμα 6.3: Μπάρα πιθανών ενεργειών του χρήστη

Στην μπάρα από αριστερά προς τα δεξιά ο χρήστης πατώντας τα κουμπιά μπορεί:

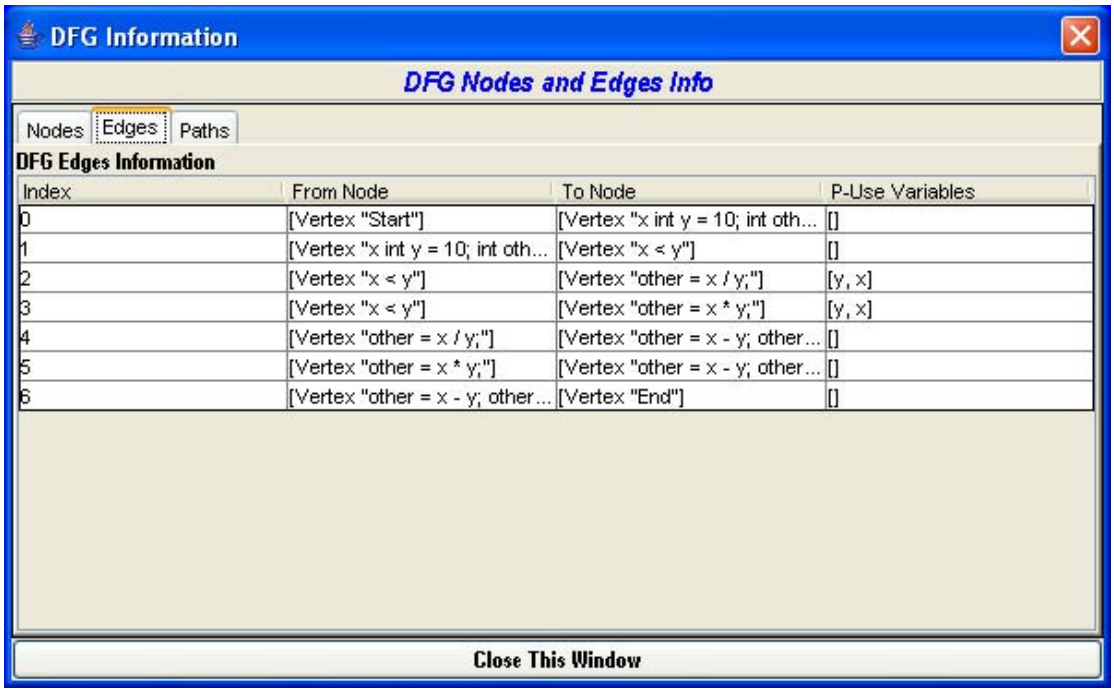
- Να επιλέξει το αρχείο που επιθυμεί,
- Να δει τον πηγαίο κώδικα σε ειδικό χώρο στα δεξιά του κυρίως παραθύρου
- Να δει τον γράφο ροής δεδομένων,
- Να δει τον γράφο ροής ελέγχου,
- Να δει τα στοιχεία ανάλυσης ροής δεδομένων (μονοπάτια, ακμές και κόμβους)
- Να προχωρήσει σε αυτόματο έλεγχο του κώδικα,
- Να δει τα αποτελέσματα του ελέγχου
- Να επανεκκινήσει το πρόγραμμα και
- Να κλείσει την εφαρμογή.

Οι τέσσερις πρώτες επιλογές είναι αρκετά ξεκάθαρες. Στην πέμπτη επιλογή, ο χρήστης μπορεί να δει όλα τα απαραίτητα στοιχεία που απαιτούνται για να εφαρμοστούν τα κριτήρια κάλυψης των γράφων ροής δεδομένων. Συγκεκριμένα στο νέο παράθυρο που ανοίγει, ο χρήστης μπορεί να δει 3 διαφορετικά tabs. Στο πρώτο tab μπορεί να δει πληροφορίες για τους κόμβους (σχήμα 6.4). Συγκεκριμένα βλέπει το όνομα, τα c-use που υπάρχουν και τα defs σε κάθε κόμβο. Το όνομα κάθε κόμβου αποτελείται από τις εντολές που περιλαμβάνει. Κάθε εντολή χωρίζεται από την επόμενη με ένα ελληνικό ερωτηματικό (;). Το όνομα του κόμβου δεν μπορεί να χρησιμοποιηθεί ώς μοναδικό χαρακτηριστικό αφού 2 κόμβοι μπορεί να περιλαμβάνουν τις ίδιες εντολές αλλά σε διαφορετικό σημείο στο πρόγραμμα. Για αυτό το λόγο υπάρχει και το index κάθε κόμβου.



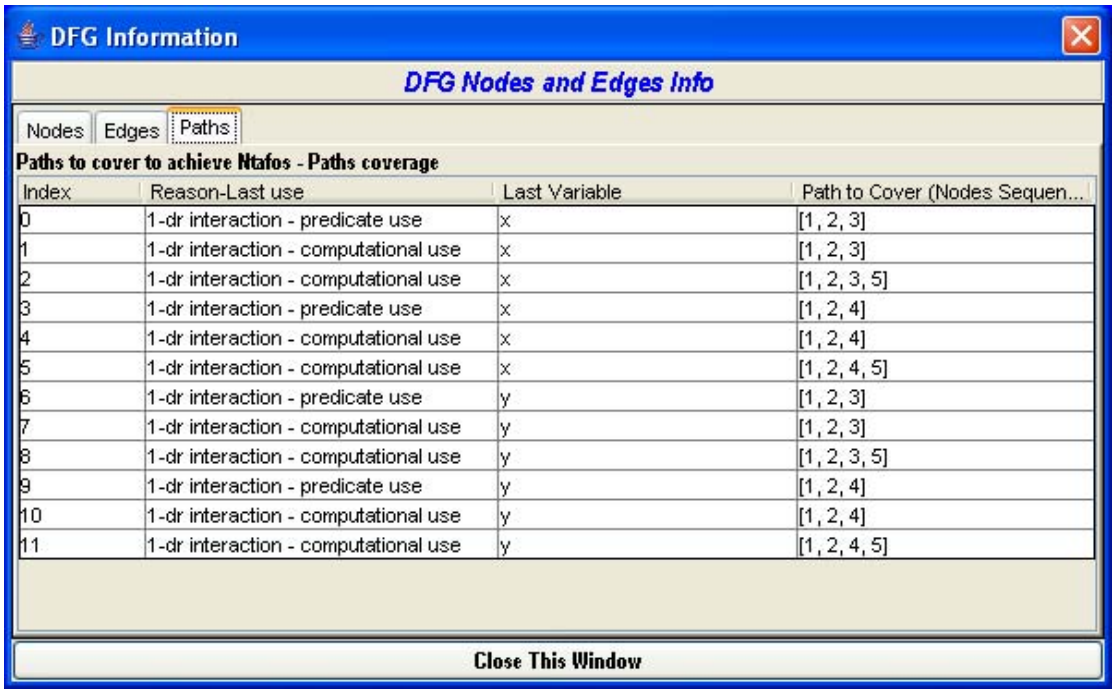
Σχήμα 6.4: Πληροφορίες για τους κόμβους

Στο δεύτερο tab μπορεί να δει πληροφορίες για τις ακμές (σχήμα 6.5). Μπορεί να δει από ποιο κόμβο ξεκινά κάθε ακμή, σε ποιο κόμβο καταλήγει και τα p-use για κάθε ακμή. Κάθε ακμή έχει διαφορετική αρχή και τέλος, σύμφωνα με τους περιορισμούς που παρουσιάστηκαν στο κεφάλαιο 3 (παράγραφος 3.2.4).



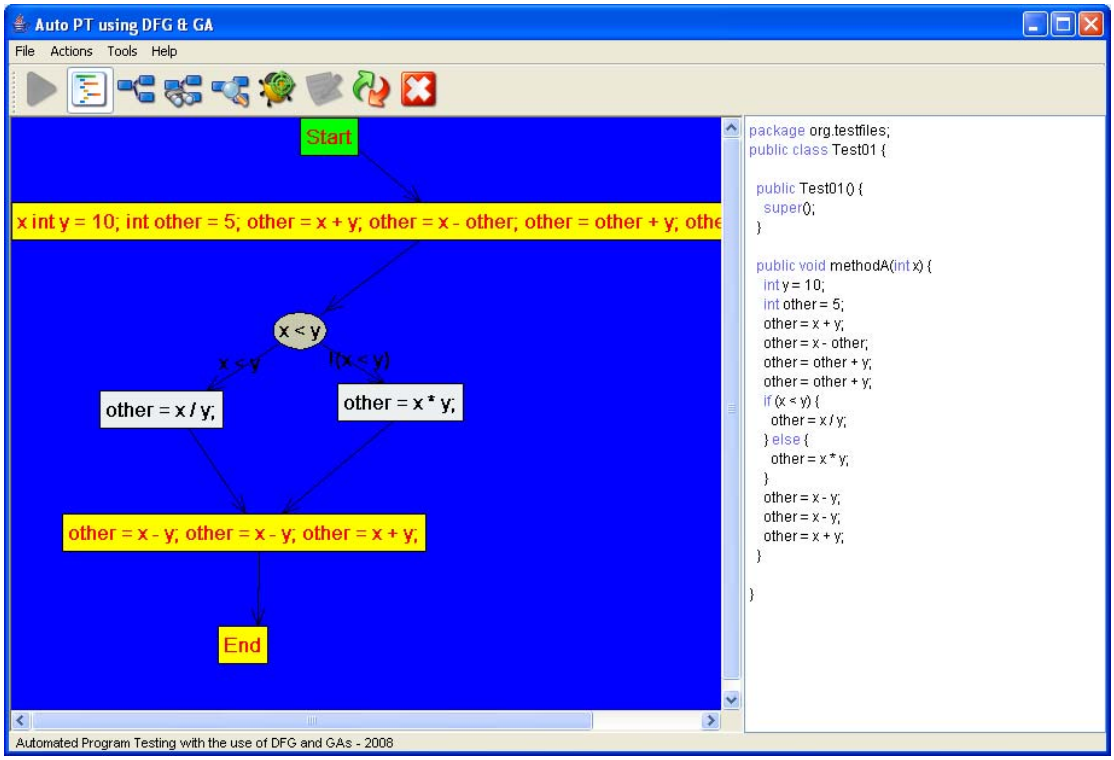
Σχήμα 6.5: Πληροφορίες για τις ακμές

Στο τρίτο και τελευταίο tab (σχήμα 6.6) ο χρήστης μπορεί να δει αναλυτικές πληροφορίες για τα μονοπάτια που πρέπει να καλυφθούν ώστε να ικανοποιηθεί το κριτήριο του Ntafos. Μπορεί να δει το είδος της τελευταίας χρήσης (c-use ή p-use) η οποία «ευθύνεται» για το μονοπάτι που πρέπει να καλυφθεί καθώς και το ποια μεταβλητή χρησιμοποιείται στην τελευταία χρήση.



Σχήμα 6.6: Τα μονοπάτια που πρέπει να καλυφθούν

Η επόμενη εικόνα (σχήμα 6.7) παρουσιάζει τα αρχικά αποτελέσματα χρήσης της αρχιτεκτονικής ανάλυσης προγραμμάτων. Είναι στο βήμα ακριβώς πριν τον έλεγχο και τη χρήση των γενετικών αλγορίθμων. Στην πιο κάτω εικόνα παρουσιάζονται οι δυνατότητες της εφαρμογής. Στο αριστερό μέρος ο χρήστης μπορεί να δει τον γράφο ροής δεδομένων που έχει δημιουργηθεί για το πρόγραμμα το οποίο εμφανίζεται στα δεξιά. Ο χρήστης της εφαρμογής έχει την δυνατότητα να «κρύψει» τον κώδικα, αυξάνοντας έτσι τον χώρο εργασίας για την αναπαράσταση του γράφου.



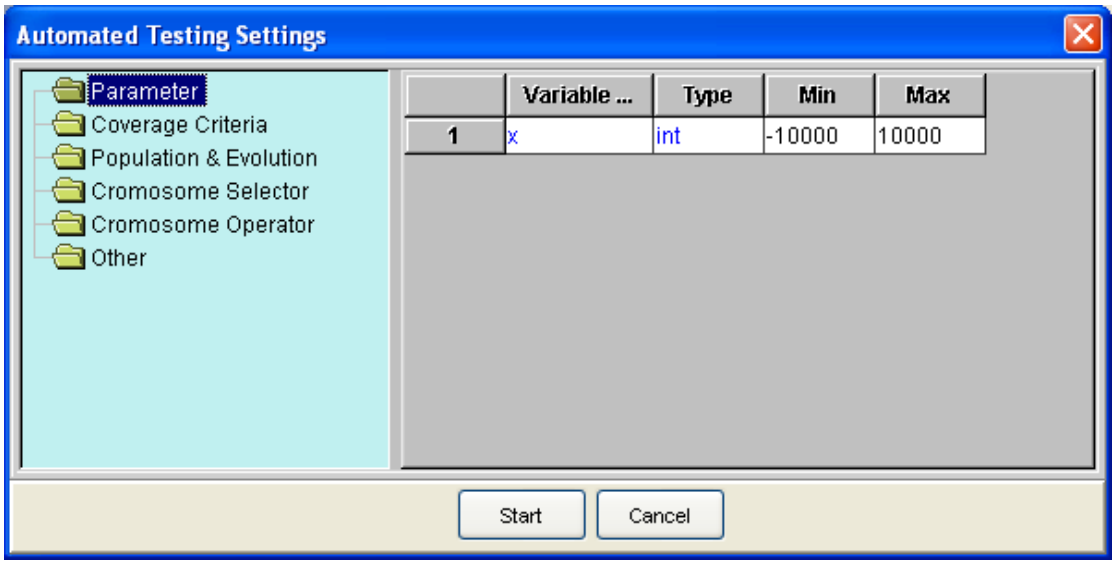
Σχήμα 6.7: Η διεπαφή χρήστη (GUI) που έχει αναπτυχθεί

Στο σημείο αυτό θα παρουσιάσουμε την διεπαφή από την στιγμή που ο χρήστης θέτει τις παραμέτρους του γενετικού αλγόριθμου μέχρι το σημείο που μπορεί να δει τα αποτελέσματα του ελέγχου.

Αφού ο χρήστης έχει επιλέξει το πρόγραμμα το οποίο θέλει να ελέγξει, και έχει δημιουργήσει το γράφο ροής δεδομένων πατώντας το ειδικό κουμπί (), τότε μπορεί να προχωρήσει στον έλεγχο. Αυτό γίνεται πατώντας στο 6ο κουμπί της μπάρας επιλογών (). Τότε εμφανίζεται στον χρήστη το παράθυρο παραμέτρων του γενετικού αλγορίθμου με 6 διαφορετικές κατηγορίες επιλογών.

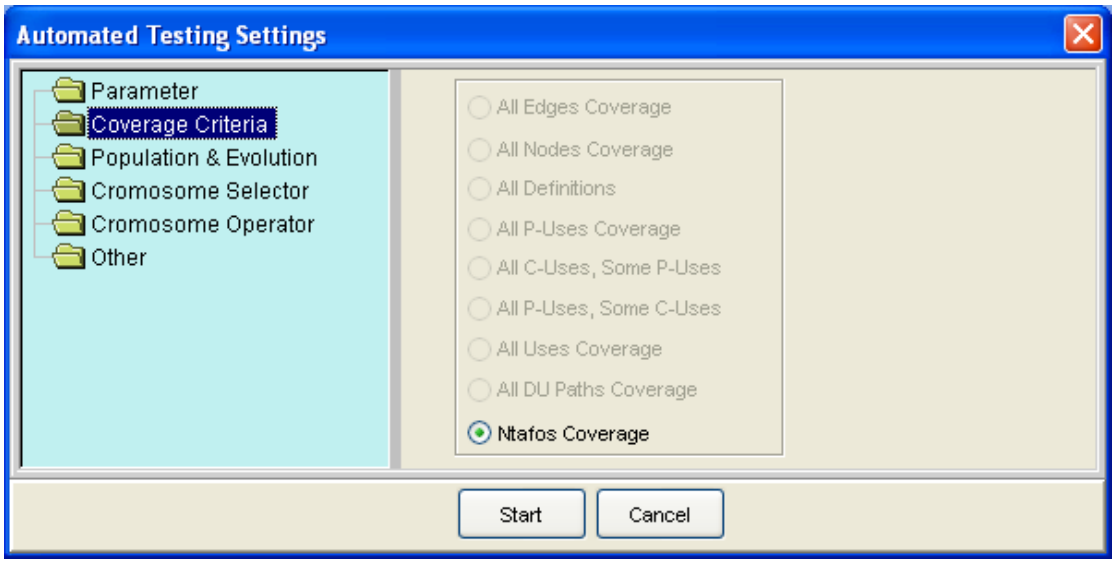
Στην πρώτη κατηγορία κανονικά ο χρήστης θα μπορούσε να καθορίσει το πεδίο τιμών των μεταβλητών εισόδου. Εδώ όμως έγινε μια τροποποίηση και αγνοείται το min και το max που δίνει ο χρήστης. Το πρόγραμμα προσπαθεί να βρει test cases δυναμικά

ανάλογα με τον αριθμό του evolution που βρίσκεται. Και αφού τα evolutions ξεκινούν από το ένα και προχωράνε μέχρι το max evolution έτσι και τα test cases που παράγονται ξεκινούν από χαμηλά και ανάλογα με μια τυχαιότητα παίρνουν θετική ή αρνητική τιμή. Αυτό η δυναμική παραγωγή test cases βοηθά περισσότερο το πρόγραμμα να καλύψει τα μονοπάτια γι' αυτό άλλωστε έγινε και η τροποποίηση. Το πρόγραμμα είναι επίσης σε θέση να βρίσκει το πλήθος και είδος των μεταβλητών που χρειάζονται για να λειτουργήσει το πρόγραμμα.



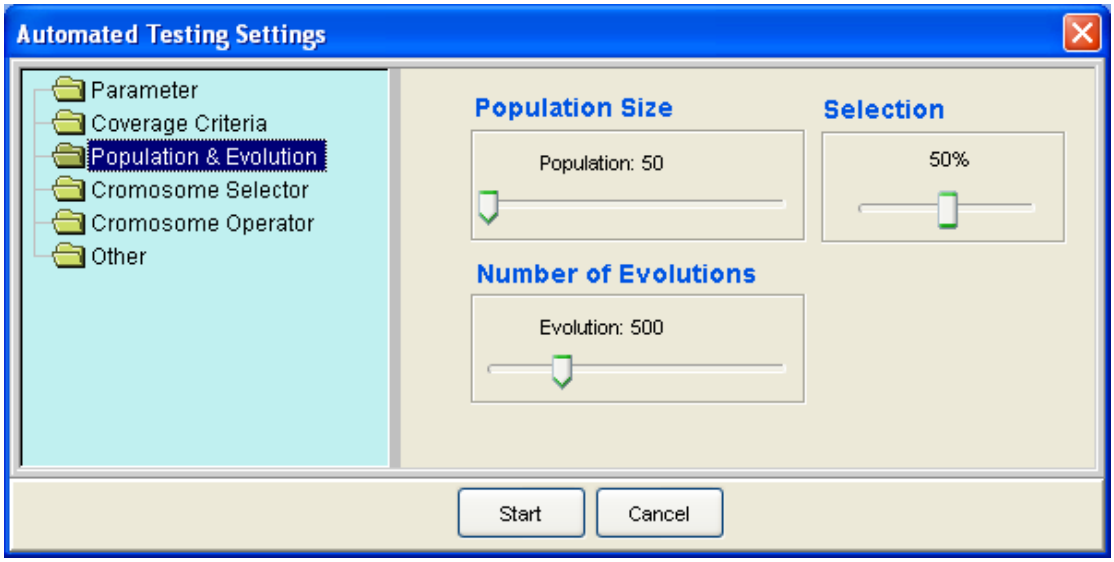
Σχήμα 6.8: Κατηγορία 1 – Καθορισμός ορίων

Στην δεύτερη κατηγορία ο χρήστης επιλέγει το κριτήριο με βάση το οποίο θα ελεγχθεί το πρόγραμμα. Στην παρούσα εργασία υλοποιήθηκε το κριτήριο Ntafos (Required k-tubles). Εδώ παρουσιάζονται τα κριτήρια γράφων ροής δεδομένων επειδή στο πρόγραμμα ορίστηκε ότι ο έλεγχος γίνεται βάσει αυτού του είδους γράφων.



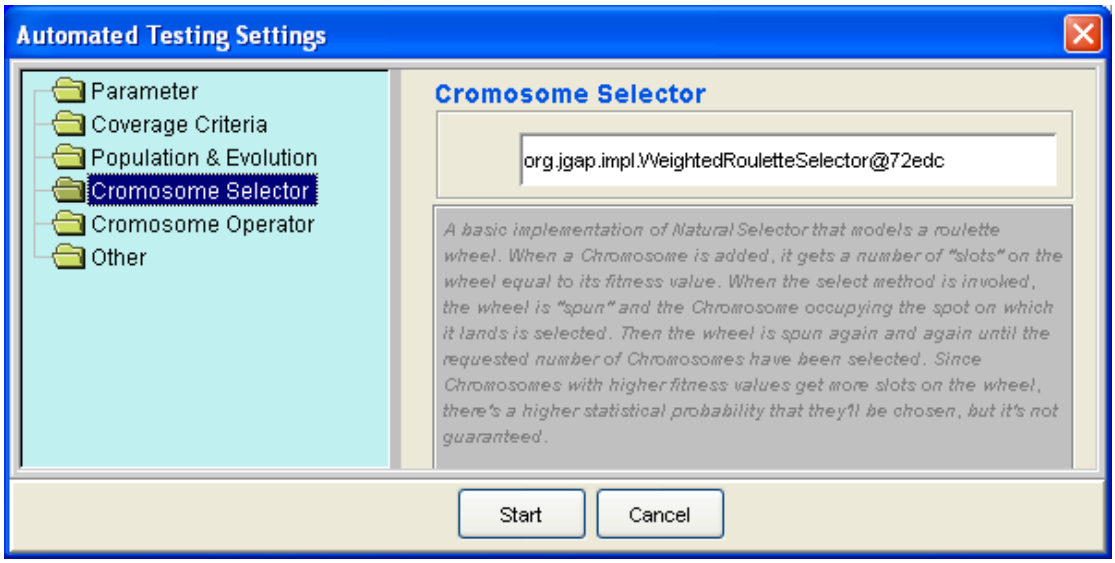
Εικόνα 6.9: Κατηγορία 2 – Επιλογή κριτηρίου κάλυψης

Στην τρίτη κατηγορία επιλογών ο χρήστης μπορεί να θέσει το μέγεθος του πληθυσμού, τον μέγιστο αριθμό evolutions και το ποσοστό από χρωματοσώματα τα οποία θα περάσουν στην επόμενη φάση (εννοείται ότι περνούν τα υγιέστερα). Περισσότερες εξελίξεις και μεγαλύτερος πληθυσμός δεν εγγυώνται απαραίτητα και καλύτερη λύση. Οι καλύτερες ρυθμίσεις για τις επιλογές αυτές είναι κάτι που μπορεί να βρεθεί εμπειρικά.



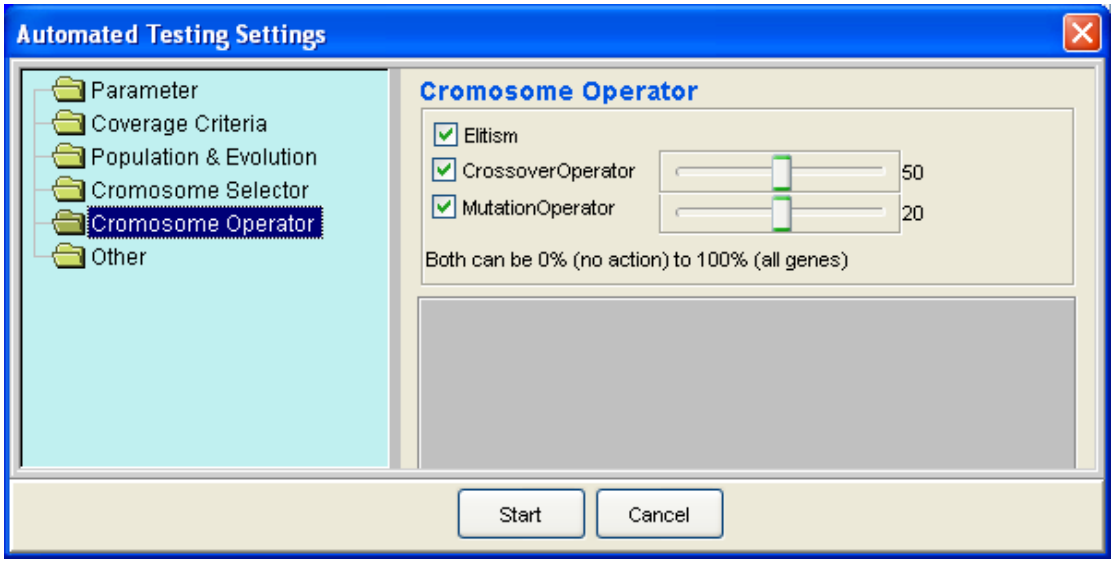
Εικόνα 6.10: Κατηγορία 3 – Καθορισμός μεγέθους πληθυσμού, evolutions και ποσοστού επιλογής

Στην τέταρτη κατηγορία, ο χρήστης μπορεί να θέσει τον τελεστή επιλογής χρωμοσωμάτων από τον υφιστάμενο πληθυσμό για την φάση της αναπαραγωγής. Ο τελεστής επιλογής που χρησιμοποιείται είναι η επιλογή βάση τροχού ρουλέτας.



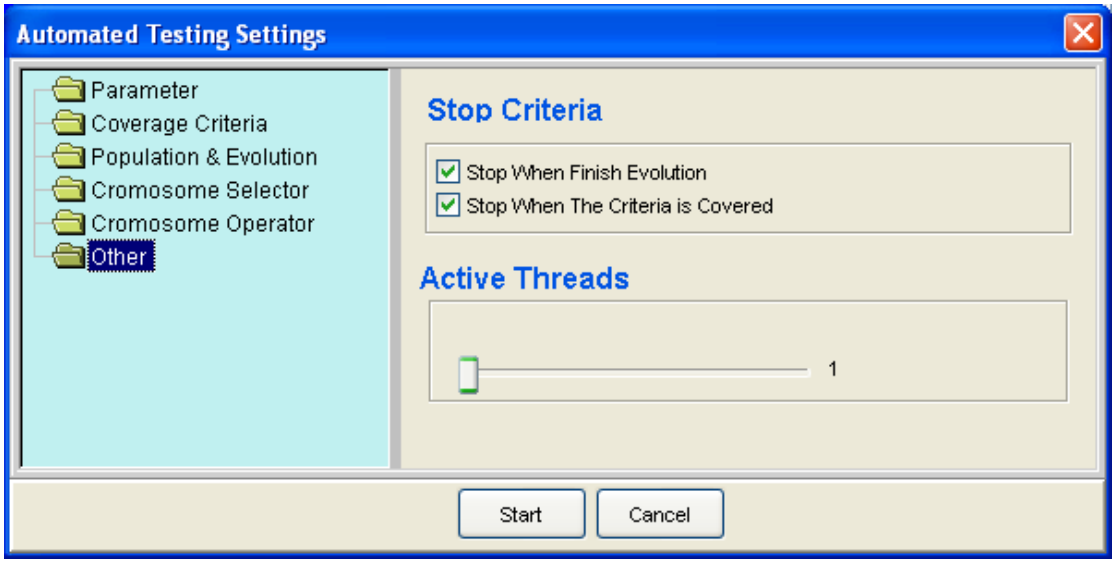
Εικόνα 6.11: Κατηγορία 4 – Τελεστής επιλογής χρωματοσωμάτων

Στην πέμπτη κατηγορία επιλογών, ο χρήστης μπορεί να επιλέξει τους τελεστές που μπορούν να χρησιμοποιηθούν για ανά-συνδυασμό των γονέων και την δημιουργία νέων χρωματοσωμάτων, καθώς και το ποσοστό που θα εφαρμοστούν. Ο χρήστης μπορεί να θέσει την επιλογή Ελιτισμού (Elitism). Αν επιλέξει ελιτισμό, τότε η καλύτερη γνωστή λύση κάθε γενιάς περνά στην επόμενη φάση ανεξάρτητα την διαδικασία επιλογής.



Εικόνα 6.12: Κατηγορία 5 – Τελεστές ανά-συνδυασμού

Τέλος στην κατηγορία «άλλες επιλογές» ο χρήστης μπορεί να εκφράσει την επιθυμία του να σταματήσει ο αλγόριθμος όταν καλυφτούν οι εξελίξεις που όρισε ή όταν βρεθεί η λύση ή και τα δύο μαζί. Το λογικό πάντως είναι να σταματά το πρόγραμμα όταν όλα τα μονοπάτια καλυφθούν. Στην παρούσα φάση μόνο ένα thread υποστηρίζεται.

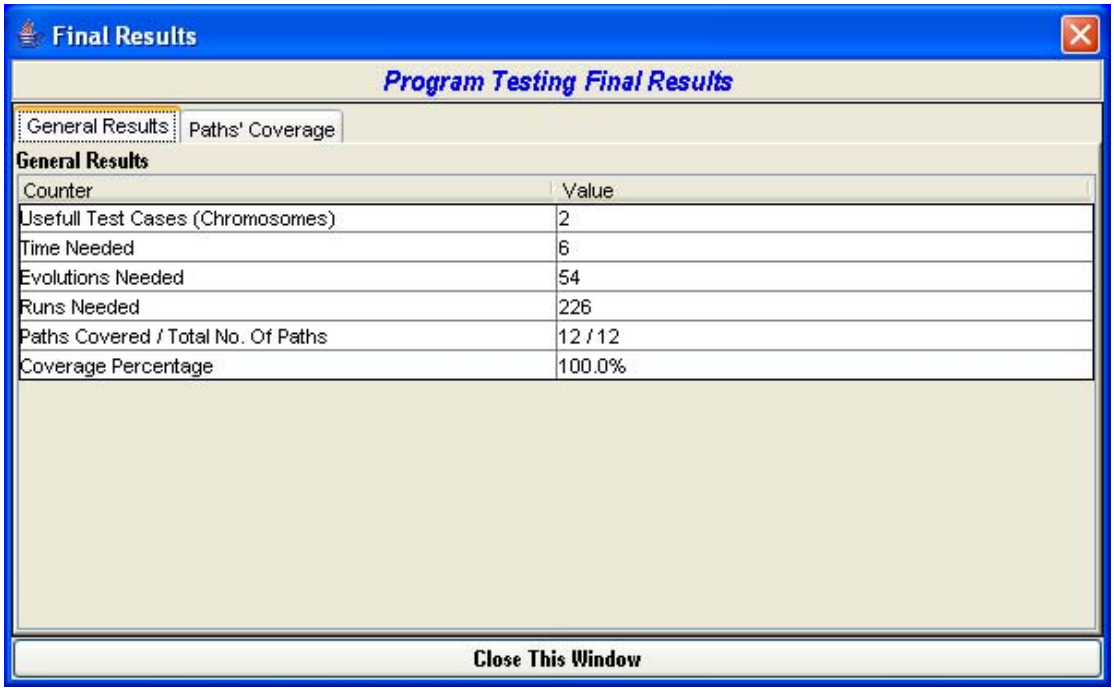


Εικόνα 6.13: Κατηγορία 6 – Κριτήρια τερματισμού

Όταν οι πιο πάνω επιλογές έχουν τεθεί, ο χρήστης μπορεί να πατήσει το κουμπί έναρξης. Τότε αρχίζει να εκτελείται ο γενετικός αλγόριθμος όπως έχει σχεδιαστεί για το πρόβλημα που εξετάζουμε και βάσει των επιλογών του χρήστη. Η εκτέλεση του αλγορίθμου αυτόματης παραγωγής δεδομένων εισόδου για έλεγχο του προγράμματος, εξαρτάται σε μεγάλο βαθμό από τις παραμέτρους που έχει ορίσει ο χρήστης. Κατά τη διάρκεια της εκτέλεσης, ο χρήστης δεν μπορεί να κάνει κάτι, εκτός από το να ακυρώσει την εκτέλεση. Μετά το τέλος εκτέλεσης, ο χρήστης μπορεί να επιλέξει το 7ο κουμπί ( ) για να του παρουσιαστούν τα αποτελέσματα.

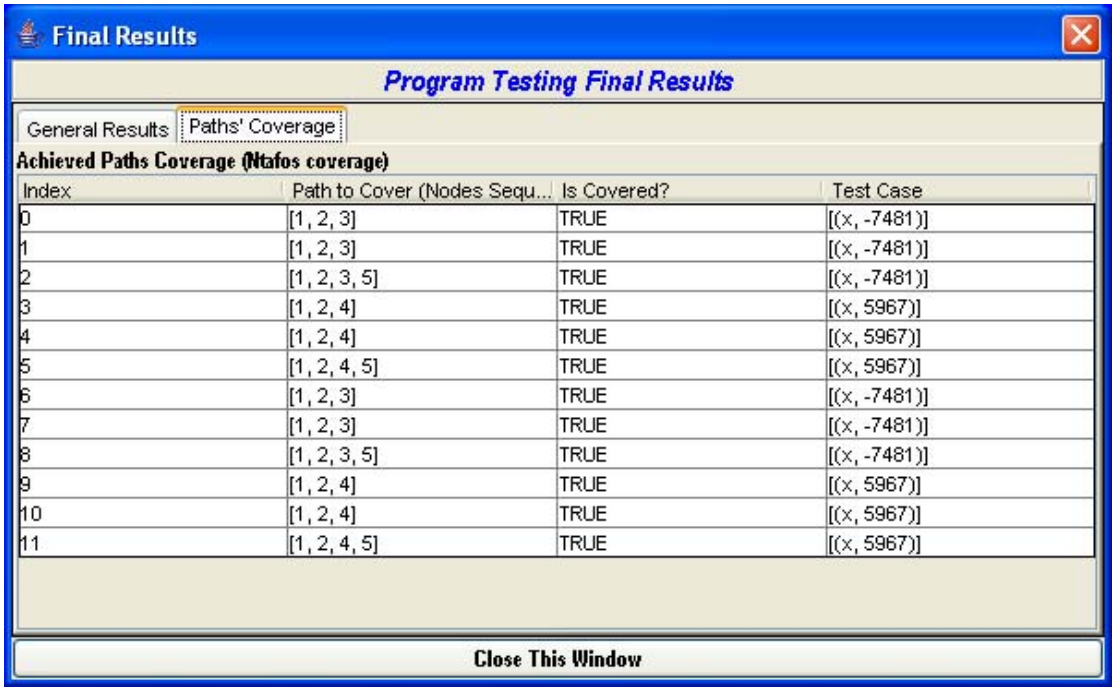
Στο νέο παράθυρο που ανοίγει, υπάρχουν 2 tabs. Στο πρώτο tab ο χρήστης μπορεί να δει γενικά στοιχεία για την εκτέλεση του αλγορίθμου. Τα στοιχεία αυτά είναι:

- Ο αριθμός των «χρήσιμων» χρωματοσωμάτων. Δηλαδή τον αριθμό χρωματοσωμάτων που χρειάζονται για να επιτευχθεί η λύση,
- Ο χρόνος που χρειάστηκε για να εκτελεστεί ο αλγόριθμος,
- Ο αριθμός των εξελίξεων που χρειάστηκε (αριθμός διαφορετικών γενεών),
- Ο αριθμός των χρωματοσωμάτων που εξετάστηκαν,
- Ο αριθμός μονοπατιών που καλύφθηκαν σε σχέση με τον συνολικό αριθμό που όριζε το κριτήριο,
- Το ποσοστό κάλυψης



Εικόνα 6.14: Γενικά αποτελέσματα για το τρέξιμο του αλγορίθμου

Στο επόμενο tab ο χρήστης μπορεί να δει την κάλυψη μονοπατιών. Συγκεκριμένα δίνεται μια λίστα από τα μονοπάτια με τους κόμβους που περιλαμβάνει το κάθε ένα. Δίπλα από κάθε μονοπάτι αναφέρετε κατά πόσο έχει καλυφθεί το μονοπάτι αυτό ή όχι. Στο τέλος για κάθε μονοπάτι αναφέρονται οι τιμές των μεταβλητών εισόδου για τις οποίες επιτυγχάνεται η κάλυψη. Οι τιμές αυτές έχουν παρθεί από το χρωματόσωμα το οποίο «ευθύνεται» για την κάλυψη του μονοπατιού.



Index	Path to Cover (Nodes Sequ...	Is Covered?	Test Case
0	[1, 2, 3]	TRUE	[(x, -7481)]
1	[1, 2, 3]	TRUE	[(x, -7481)]
2	[1, 2, 3, 5]	TRUE	[(x, -7481)]
3	[1, 2, 4]	TRUE	[(x, 5967)]
4	[1, 2, 4]	TRUE	[(x, 5967)]
5	[1, 2, 4, 5]	TRUE	[(x, 5967)]
6	[1, 2, 3]	TRUE	[(x, -7481)]
7	[1, 2, 3]	TRUE	[(x, -7481)]
8	[1, 2, 3, 5]	TRUE	[(x, -7481)]
9	[1, 2, 4]	TRUE	[(x, 5967)]
10	[1, 2, 4]	TRUE	[(x, 5967)]
11	[1, 2, 4, 5]	TRUE	[(x, 5967)]

Εικόνα 6.15: Αποτελέσματα κάλυψης μονοπατιών

Κεφάλαιο 8

Αποτελέσματα και συμπεράσματα

8.1. Εισαγωγή 120
8.2. Αποτελέσματα: Ποσοστό κάλυψης μονοπατιών του κριτηρίου Ntafos 120
8.3. Σύγκριση με προηγούμενη δουλειά 129
8.4. Αποτελέσματα και συμπεράσματα 130
8.5. Μελλοντική εργασία 134

8.1. Εισαγωγή

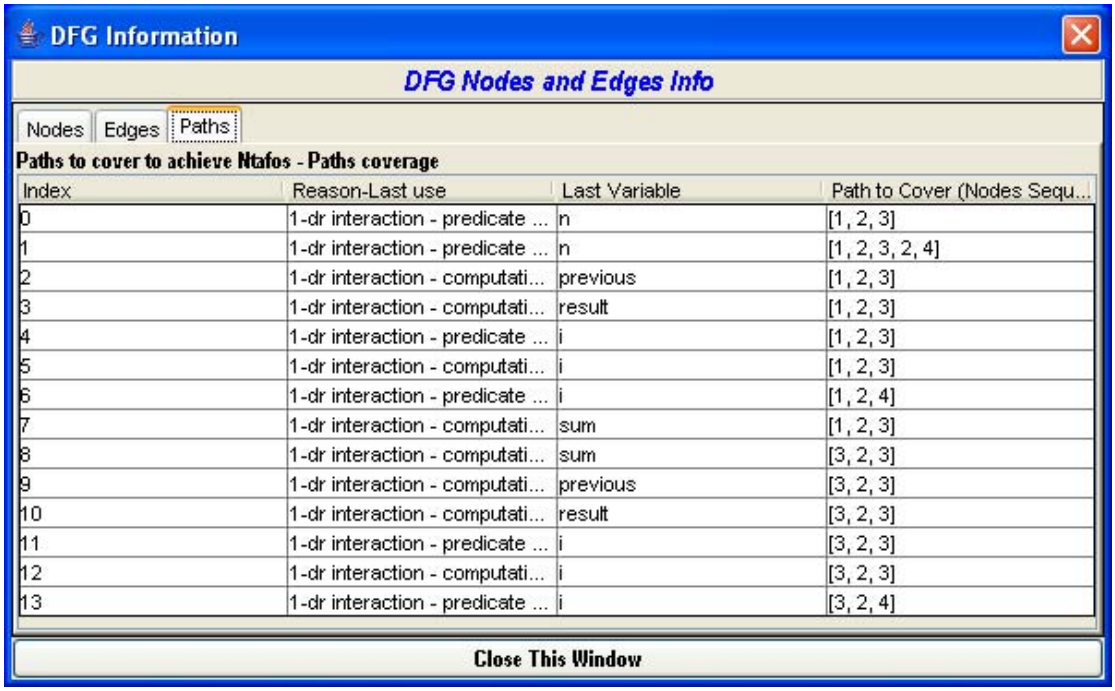
Σε αυτό το κεφάλαιο θα παρουσιαστούν τα αποτελέσματα τις εργασίας αυτής. Στην παράγραφο 8.2. θα βρεθούν τα ποσοστά κάλυψης των μονοπατιών που παράγονται με το κριτήριο του Ntafos, με γνωστά όμως προγράμματα (benchmarks programs). Στο τέλος της παραγράφου αυτής θα δοθεί ένας πίνακας που συνοψίζει τα αποτελέσματα αυτά καθώς και τα αποτελέσματα που πάρθηκαν από την προηγούμενη εργασία με το κριτήριο ALL-DU paths. Έπειτα στην παράγραφο 8.3 θα γίνει σύγκριση της παρούσας δουλειάς με την δουλειά στο [5]. Εδώ θα δείξω ότι τα μονοπάτια που παράγονται στο [5] με το κριτήριο ALL-DU paths εμπεριέχονται στο κριτήριο του Ntafos, το οποίο υλοποιήθηκε στην παρούσα εργασία. Στην παράγραφο 8.4 θα παρουσιαστούν κάποια αποτελέσματα και συμπεράσματα που πάρθηκαν μέσω κάποιων πειραμάτων. Τέλος παρουσιάζονται κάποιες σκέψεις για μελλοντική εργασία.

8.2. Αποτελέσματα: Ποσοστό κάλυψης μονοπατιών του κριτηρίου Ntafos

Τα προγράμματα που χρησιμοποιήθηκαν για να βρεθεί το ποσοστό κάλυψης των μονοπατιών είναι γνωστά προγράμματα που εξυπηρετούν συγκεκριμένο σκοπό. Τα προγράμματα αυτά είναι:

- **Fibonacci.java**: επιστρέφει τα άθροισμα της ακολουθίας n Fibonacci αριθμών.
- **FindMaximum.java**: βρίσκει και επιστρέφει μεταξύ δύο αριθμών τον μεγαλύτερο.
- **FindMinimum.java**: βρίσκει και επιστρέφει μεταξύ δύο αριθμών τον μικρότερο.
- **SumExample.java**: επιστρέφει το άθροισμα των n αριθμών που του δίδεται ως παράμετρος.

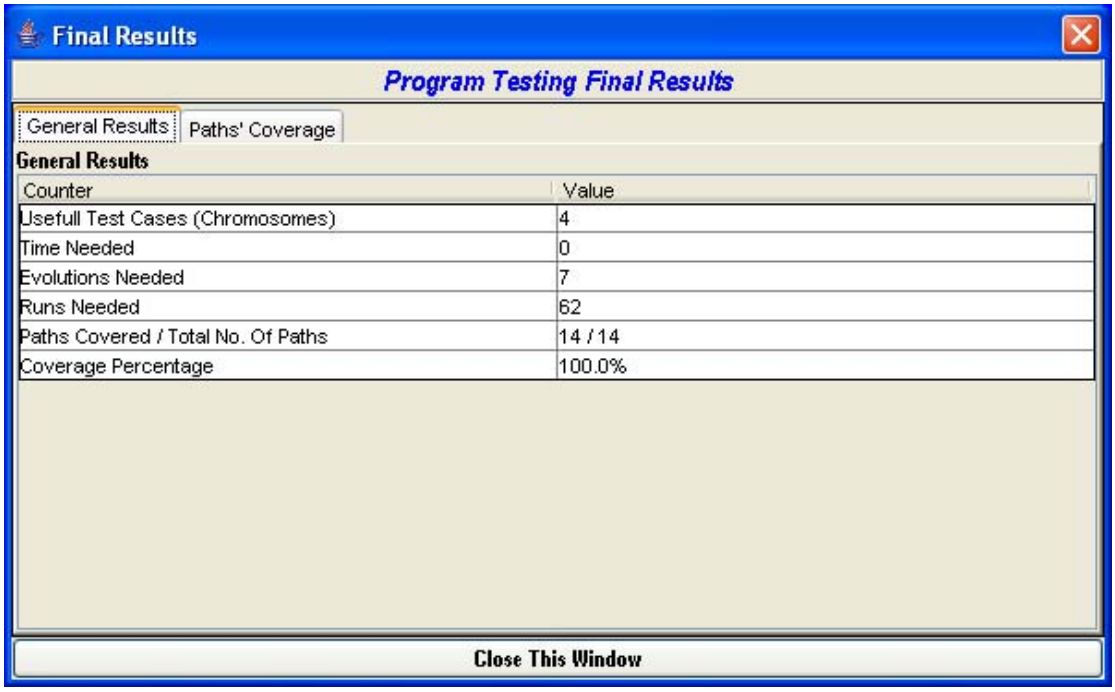
Για το πρόγραμμα **Fibonacci.java** τα μονοπάτια που παράγονται είναι τα παρακάτω:



Index	Reason-Last use	Last Variable	Path to Cover (Nodes Sequ...
0	1-dr interaction - predicate ...	n	[1, 2, 3]
1	1-dr interaction - predicate ...	n	[1, 2, 3, 2, 4]
2	1-dr interaction - computati...	previous	[1, 2, 3]
3	1-dr interaction - computati...	result	[1, 2, 3]
4	1-dr interaction - predicate ...	i	[1, 2, 3]
5	1-dr interaction - computati...	i	[1, 2, 3]
6	1-dr interaction - predicate ...	i	[1, 2, 4]
7	1-dr interaction - computati...	sum	[1, 2, 3]
8	1-dr interaction - computati...	sum	[3, 2, 3]
9	1-dr interaction - computati...	previous	[3, 2, 3]
10	1-dr interaction - computati...	result	[3, 2, 3]
11	1-dr interaction - predicate ...	i	[3, 2, 3]
12	1-dr interaction - computati...	i	[3, 2, 3]
13	1-dr interaction - predicate ...	i	[3, 2, 4]

Σχήμα 8.1: Μονοπάτια που παράγονται για το αρχείο Fibonacci.java (1-dr)

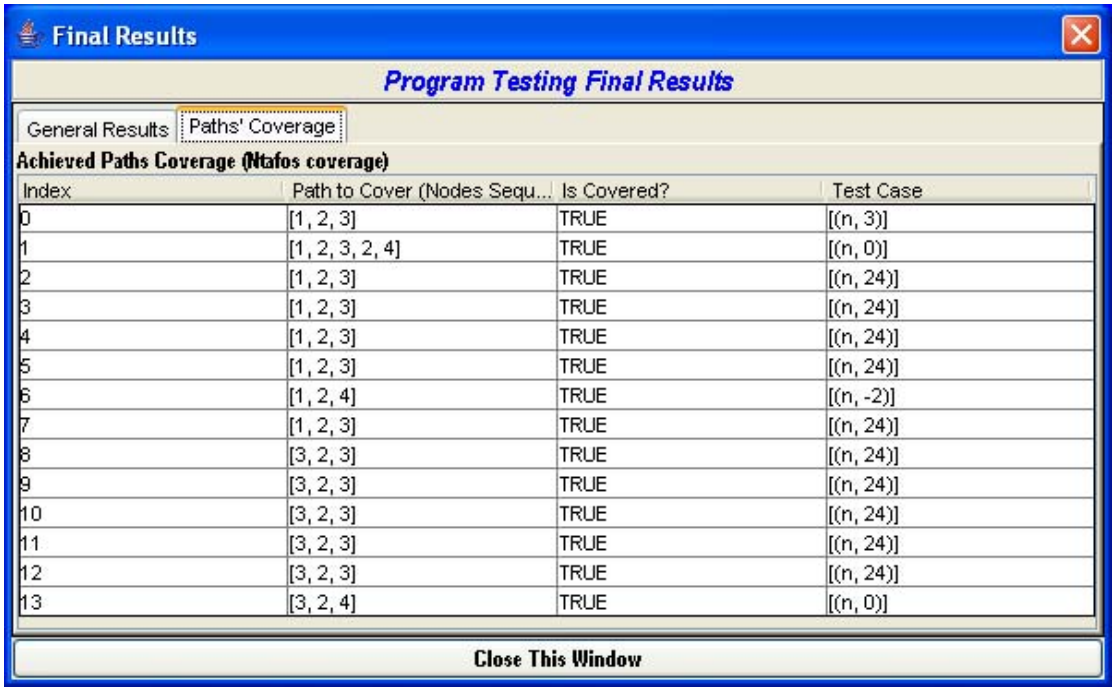
Το ποσοστό κάλυψης των μονοπατιών αυτών, σύμφωνα με την fitness function που επεξηγήθηκε στην παράγραφο 7.3.4 είναι:



Counter	Value
Usefull Test Cases (Chromosomes)	4
Time Needed	0
Evolutions Needed	7
Runs Needed	62
Paths Covered / Total No. Of Paths	14 / 14
Coverage Percentage	100.0%

Σχήμα 8.2: Κάλυψη μονοπατιών για το αρχείο Fibonacci.java (1-dr)

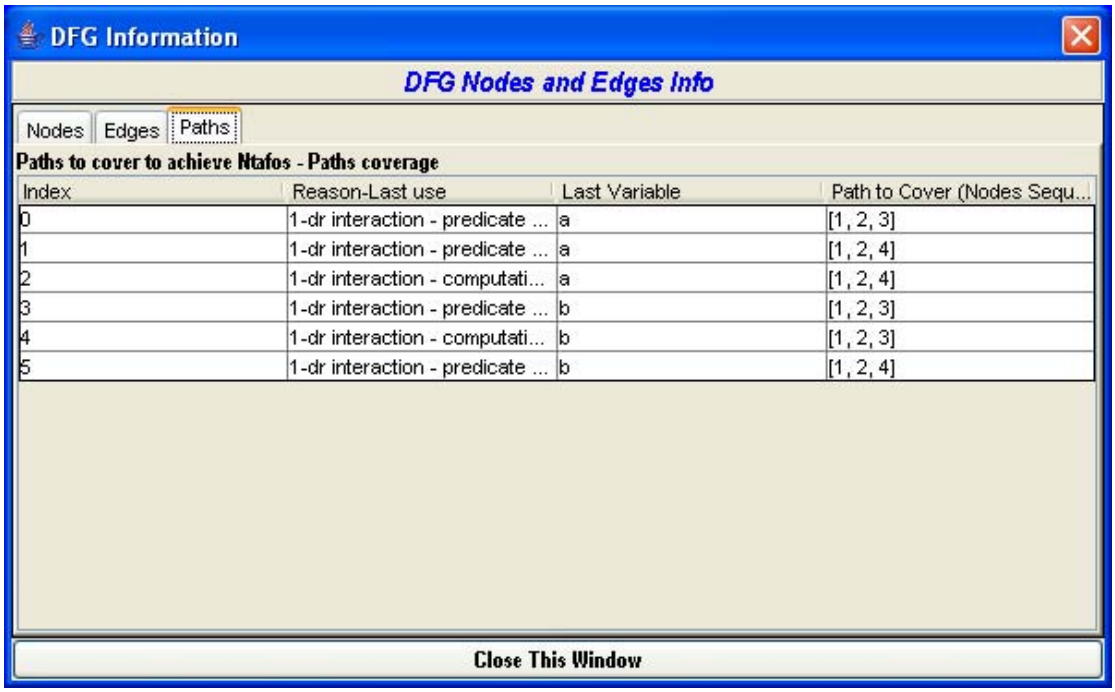
Όπως βλέπουμε το ποσοστό κάλυψης είναι 100%, αφού κάλυψε και τα 14 μονοπάτια που παραχθήκαν από το κριτήριο του Ntafos. Στην παρακάτω οθόνη βλέπουμε την κάλυψη αυτή των μονοπατιών καθώς επίσης και τα test cases (χρωμοσώματα) που ευθύνονται για την κάλυψη αυτή.



Index	Path to Cover (Nodes Sequ...	Is Covered?	Test Case
0	[1, 2, 3]	TRUE	[(n, 3)]
1	[1, 2, 3, 2, 4]	TRUE	[(n, 0)]
2	[1, 2, 3]	TRUE	[(n, 24)]
3	[1, 2, 3]	TRUE	[(n, 24)]
4	[1, 2, 3]	TRUE	[(n, 24)]
5	[1, 2, 3]	TRUE	[(n, 24)]
6	[1, 2, 4]	TRUE	[(n, -2)]
7	[1, 2, 3]	TRUE	[(n, 24)]
8	[3, 2, 3]	TRUE	[(n, 24)]
9	[3, 2, 3]	TRUE	[(n, 24)]
10	[3, 2, 3]	TRUE	[(n, 24)]
11	[3, 2, 3]	TRUE	[(n, 24)]
12	[3, 2, 3]	TRUE	[(n, 24)]
13	[3, 2, 4]	TRUE	[(n, 0)]

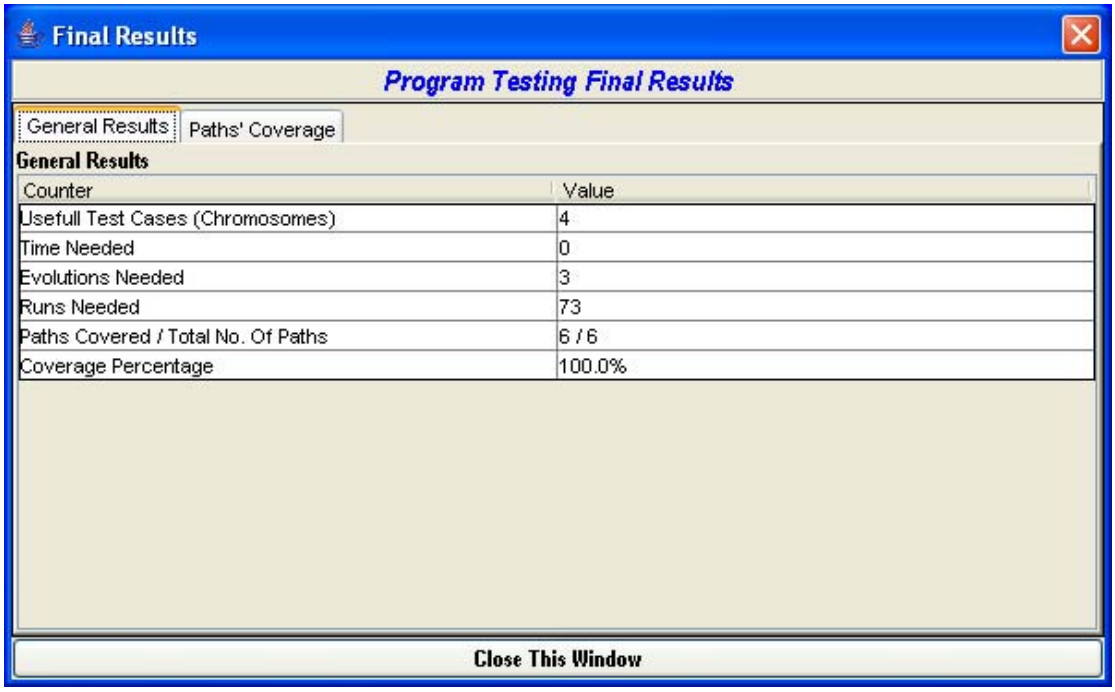
Σχήμα 8.3: Test cases που ευθύνονται για την κάλυψη των μονοπατιών του αρχείου Fibonacci.java (1-dr)

Για το πρόγραμμα **FindMaximum.java** τα μονοπάτια που παράγονται είναι τα παρακάτω:



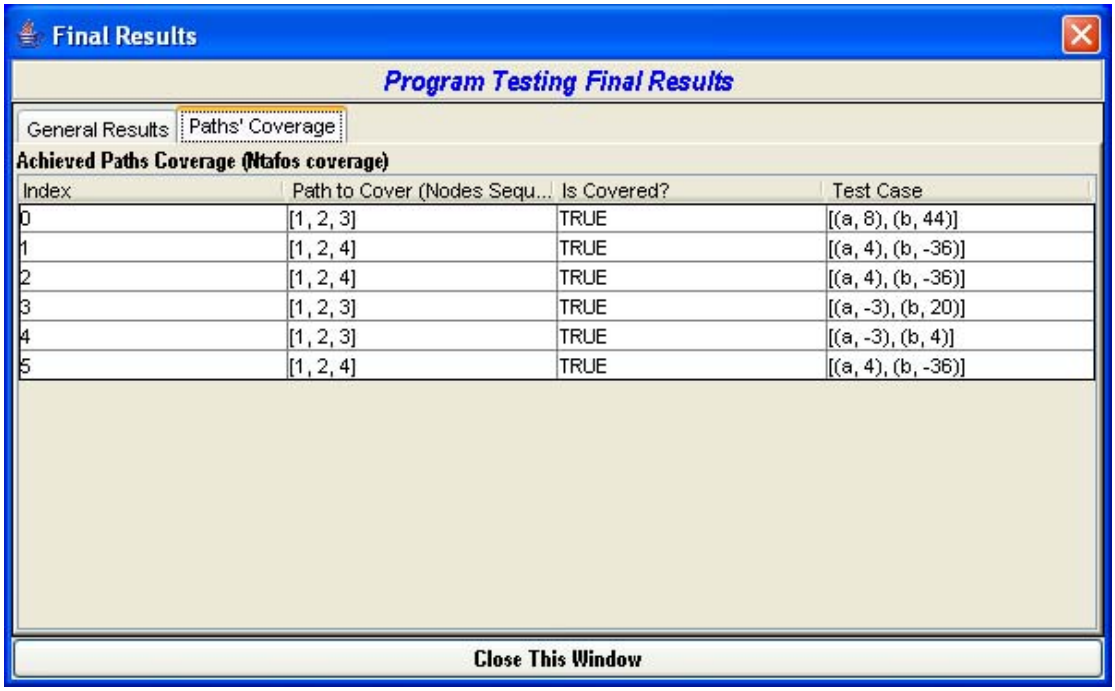
Σχήμα 8.4: Μονοπάτια που παράγονται για το αρχείο FindMaximum.java (1-dr)

Το ποσοστό κάλυψης των μονοπατιών αυτών, σύμφωνα με την fitness function που επεξηγήθηκε στην παράγραφο 7.3.4 είναι:



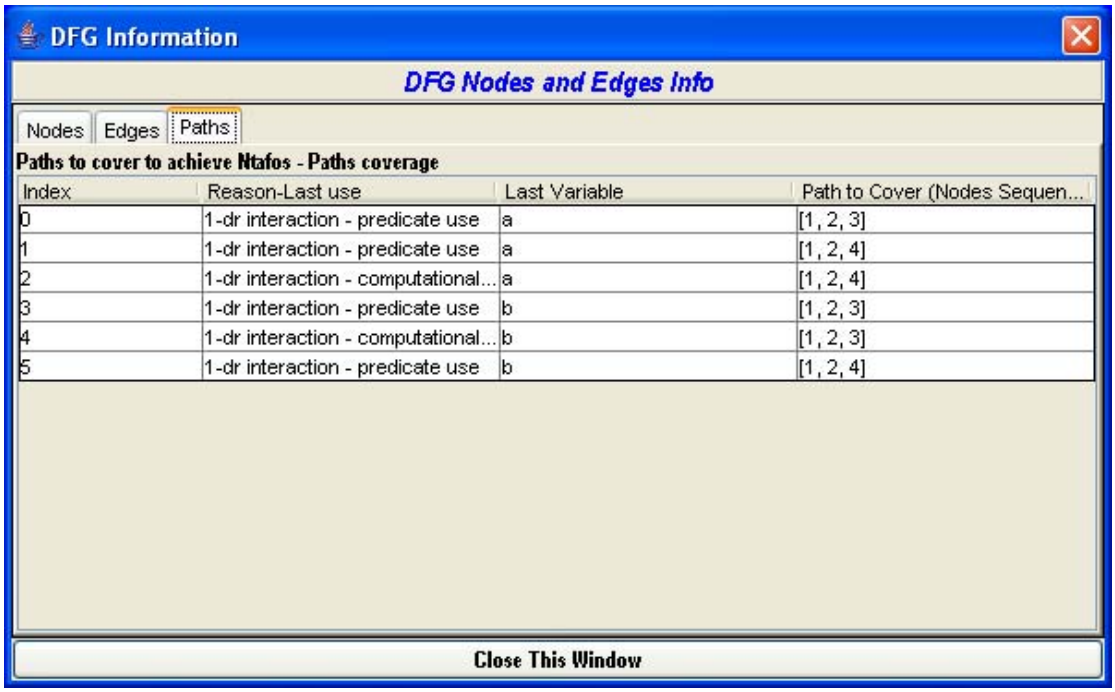
Σχήμα 8.5: Κάλυψη μονοπατιών για το αρχείο FindMaximum.java (1-dr)

Όπως βλέπουμε το ποσοστό κάλυψης είναι 100%, αφού κάλυψε και τα 6 μονοπάτια που παραχθήκαν από το κριτήριο του Ntafos. Στην παρακάτω οθόνη βλέπουμε την κάλυψη αυτή των μονοπατιών καθώς επίσης και τα test cases (χρωμοσώματα) που ευθύνονται για την κάλυψη αυτή.



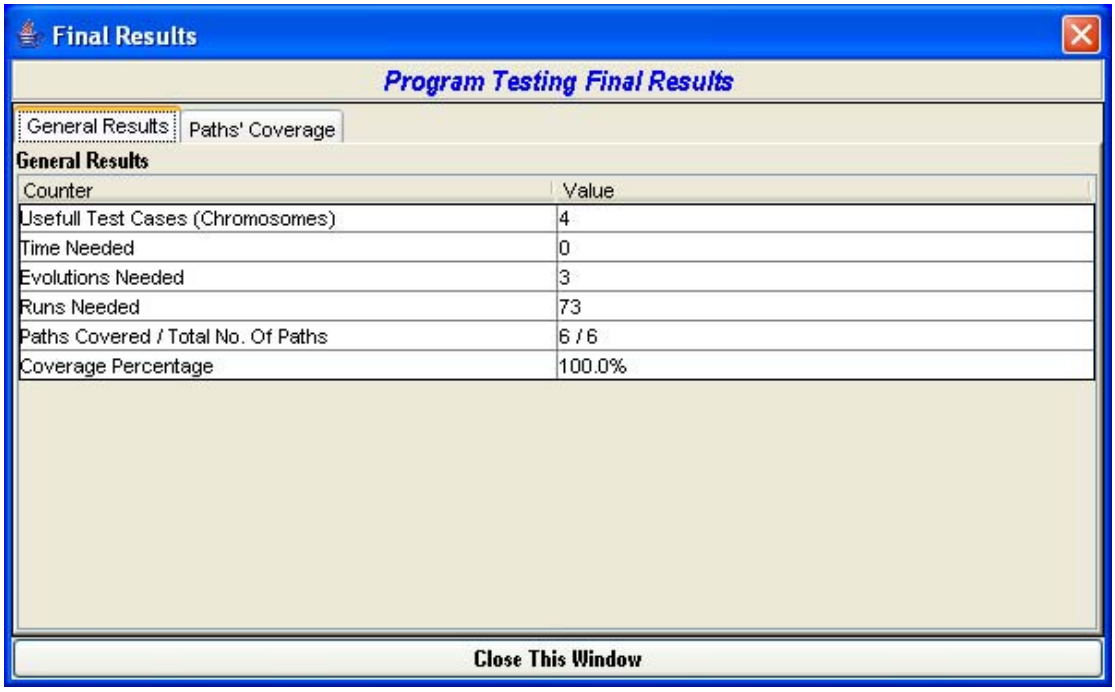
Σχήμα 8.6: Test cases που ευθύνονται για την κάλυψη των μονοπατιών του αρχείου FindMaximum.java (1-dr)

Για το πρόγραμμα **FindMinimum.java** τα μονοπάτια που παράγονται είναι τα παρακάτω:



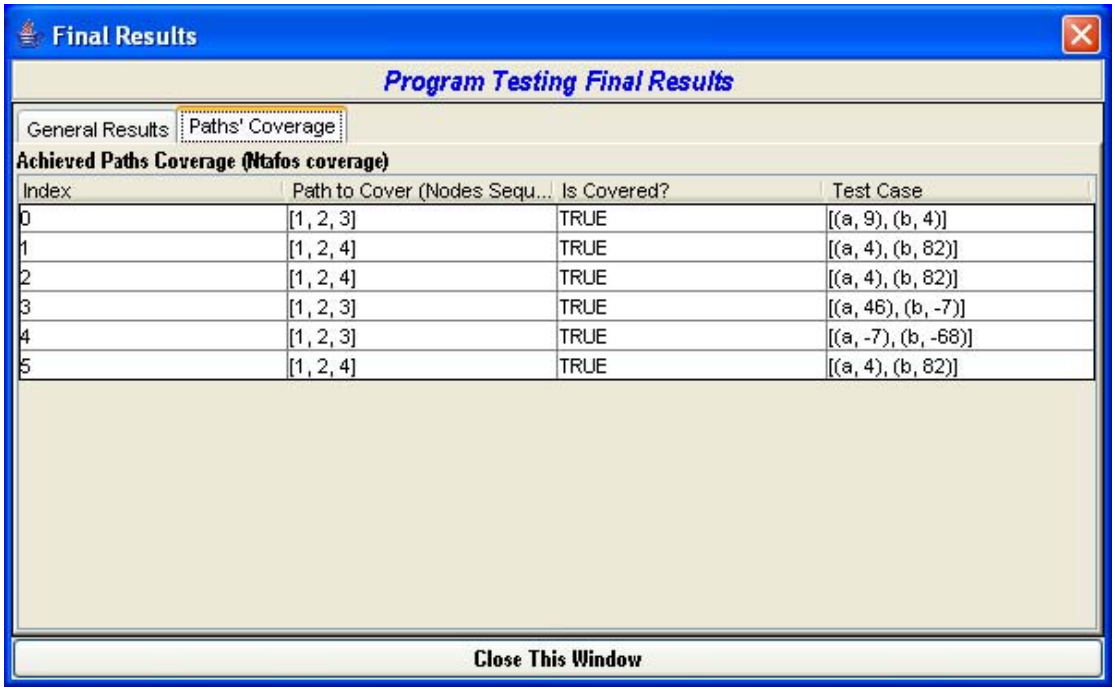
Σχήμα 8.7: Μονοπάτια που παράγονται για το αρχείο FindMinimum.java (1-dr)

Το ποσοστό κάλυψης των μονοπατιών αυτών, σύμφωνα με την fitness function που επεξηγήθηκε στην παράγραφο 7.3.4 είναι:



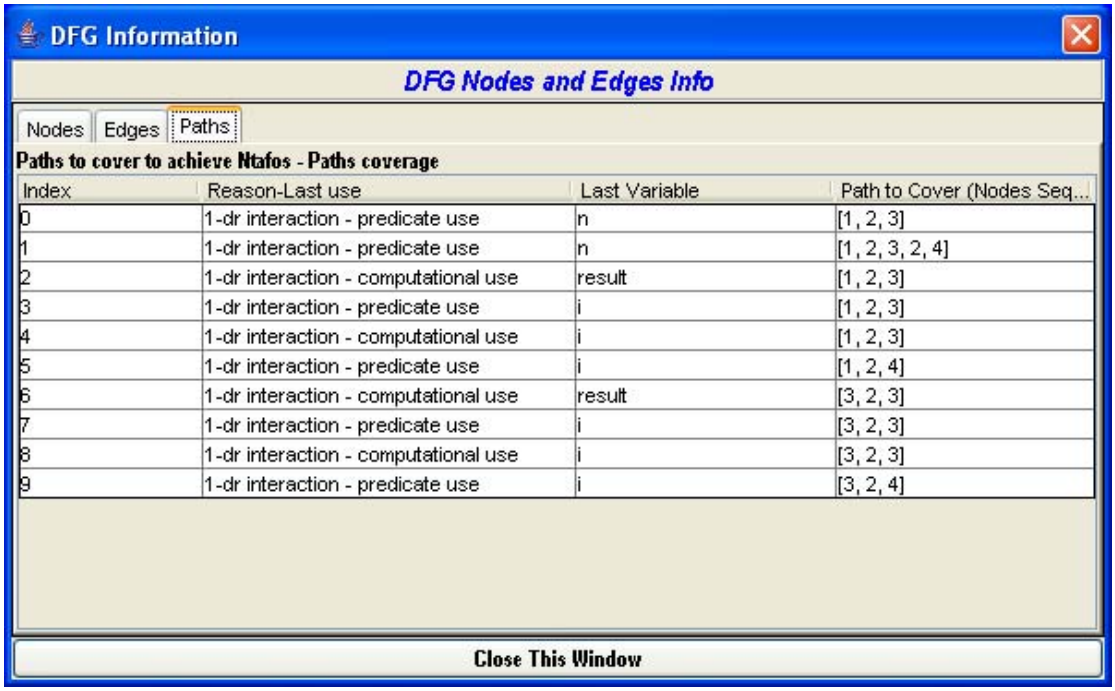
Σχήμα 8.8: Κάλυψη μονοπατιών για το αρχείο FindMinimum.java (1-dr)

Όπως βλέπουμε το ποσοστό κάλυψης είναι 100%, αφού κάλυψε και τα 6 μονοπάτια που παραχθήκαν από το κριτήριο του Ntafos. Στην παρακάτω οθόνη βλέπουμε την κάλυψη αυτή των μονοπατιών καθώς επίσης και τα test cases (χρωμοσώματα) που ευθύνονται για την κάλυψη αυτή.



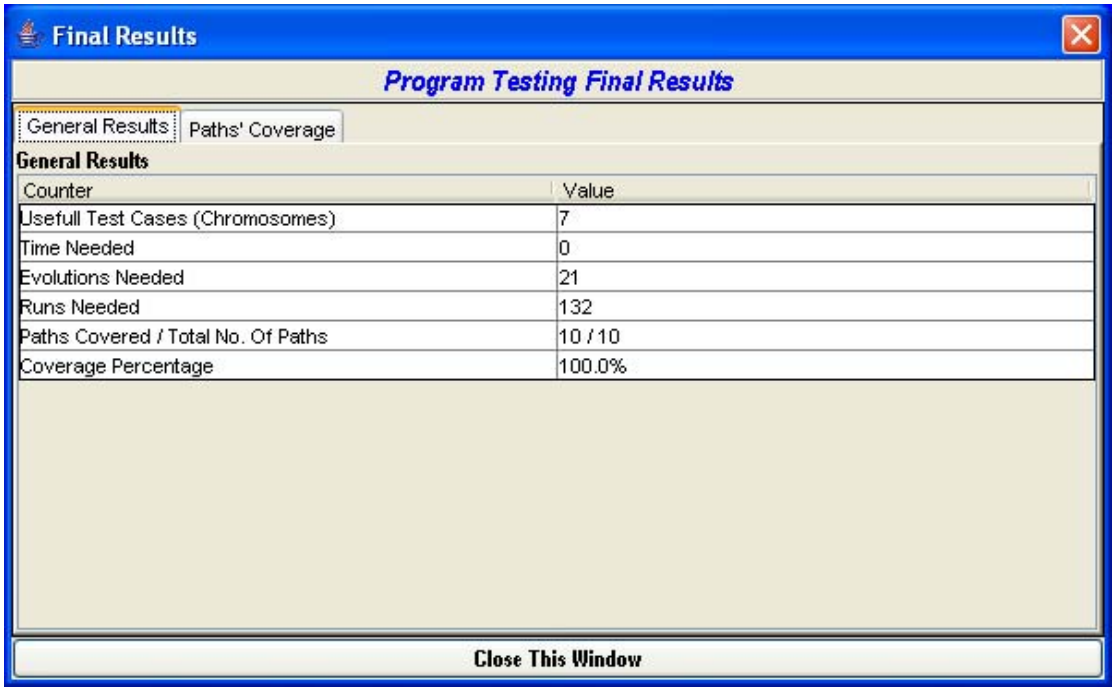
Σχήμα 8.9: Test cases που ευθύνονται για την κάλυψη των μονοπατιών του αρχείου FindMinimum.java (1-dr)

Για το πρόγραμμα **SumExample.java** τα μονοπάτια που παράγονται είναι τα παρακάτω:



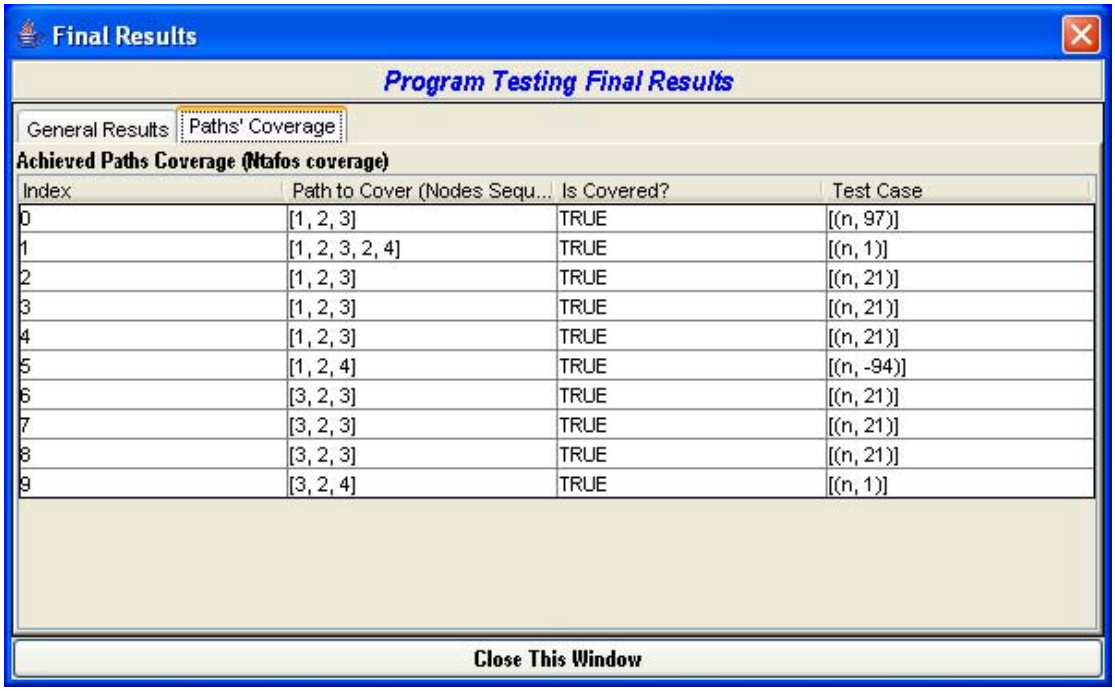
Σχήμα 8.10: Μονοπάτια που παράγονται για το αρχείο SumExample.java (1-dr)

Το ποσοστό κάλυψης των μονοπατιών αυτών, σύμφωνα με την fitness function που επεξηγήθηκε στην παράγραφο 7.3.4 είναι:



Σχήμα 8.11: Κάλυψη μονοπατιών για το αρχείο SumExample.java (1-dr)

Όπως βλέπουμε το ποσοστό κάλυψης είναι 100%, αφού κάλυψε και τα 10 μονοπάτια που παραχθήκαν από το κριτήριο του Ntafos. Στην παρακάτω οθόνη βλέπουμε την κάλυψη αυτή των μονοπατιών καθώς επίσης και τα test cases (χρωμοσώματα) που ευθύνονται για την κάλυψη αυτή.



Σχήμα 8.12: Test cases που ευθύνονται για την κάλυψη των μονοπατιών του αρχείου SumExample.java (1-dr)

Τα αποτελέσματα παρουσιάζονται συγκεντρωμένα στον πιο κάτω πίνακα:

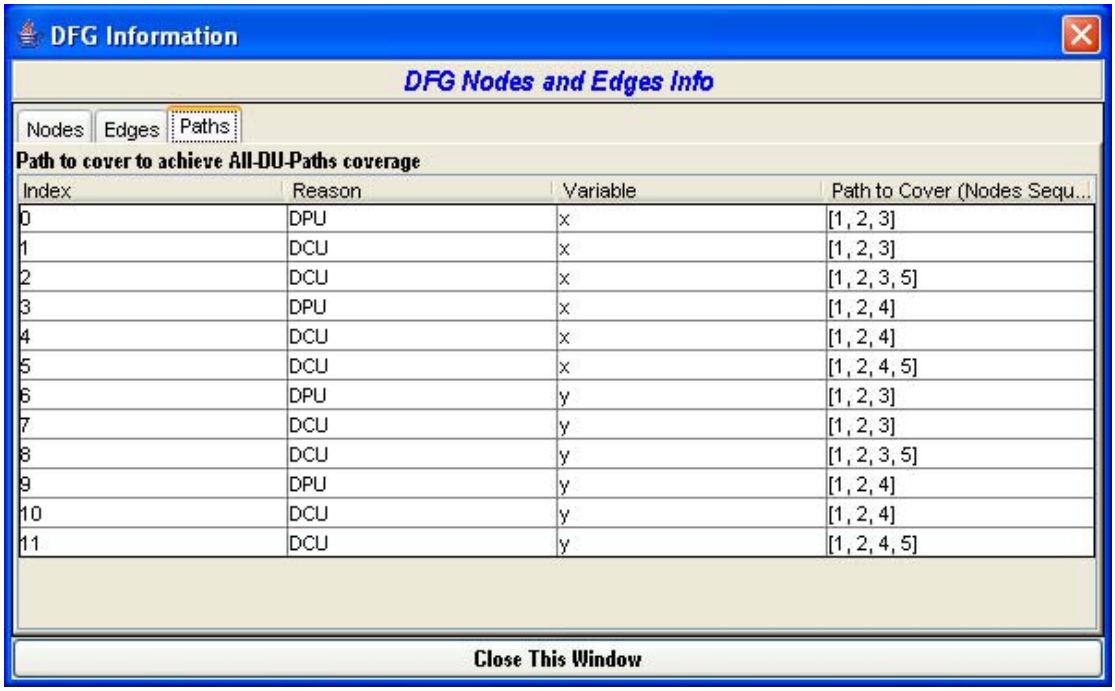
Όνομα Προγράμματος	**Κριτήριο του Ntafos (% κάλυψη)**		**All-Du Paths (% κάλυψη)**
	1-dr	**2-dr**	
Fibonacci	100%	100%	83%
FindMaximum	100%	100%	100%
FindMinimum	100%	100%	100%
SumExample	100%	92%	91%

8.3. Σύγκριση με προηγούμενη δουλειά

Σ' αυτή εδώ την παράγραφο θα κάνω μια σύγκριση της προηγούμενης δουλειάς με τη δική μου. Πιο συγκεκριμένα θα συγκρίνω το κριτήριο ALL-DU Paths που υλοποιήθηκε στο [5] μαζί με το κριτήριο του Ntafos, που υλοποιήθηκε στην παρούσα εργασία.

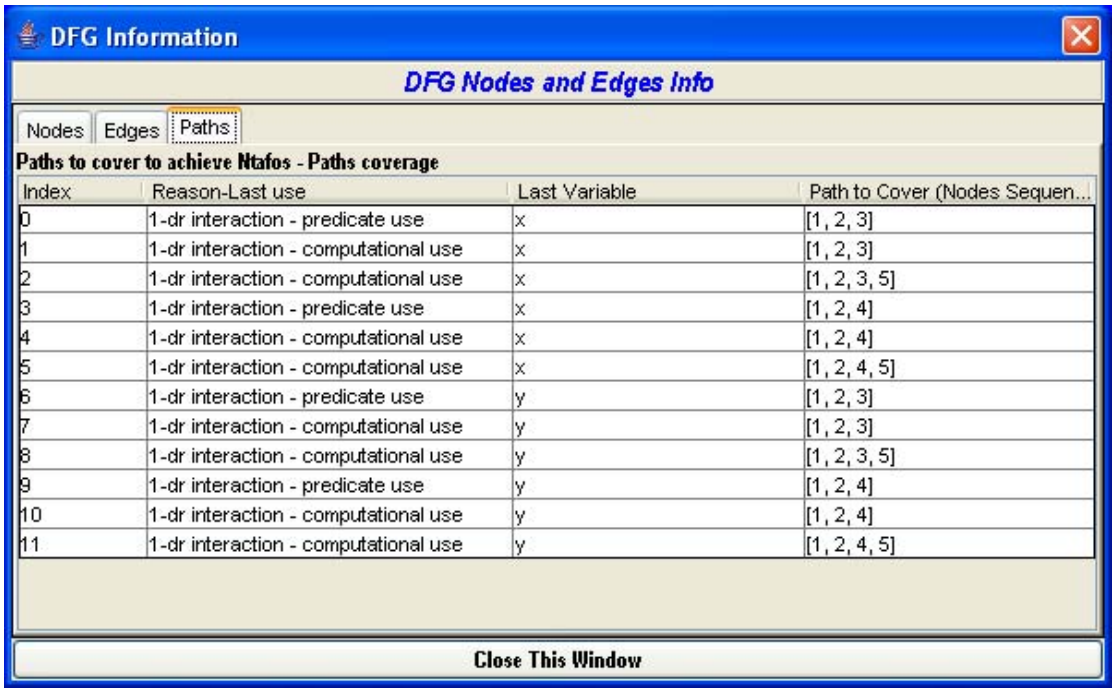
Αυτό που έχω προσέξει είναι ότι τα μονοπάτια που παράγονται με το κριτήριο ALL-DU Paths εμπεριέχονται στο κριτήριο του Ntafos για 1-dr. Στην περίπτωση που το κριτήριο του Ntafos είναι για μεγαλύτερου βαθμού (dr) μονοπάτια, πάλι τα μονοπάτια που παράγονται με το κριτήριο ALL-DU Paths εμπεριέχονται στο κριτήριο του Ntafos συν κάποια άλλα μονοπάτια, τα οποία αντιστοιχούν σε μεγαλύτερου αριθμού dr interaction. Αυτό συμβαίνει για τον λόγο ότι και στα δύο κριτήρια ξεκινούμε από κάποιο definition κάποιας μεταβλητής και αναζητούμε κάποιο use (είτε p-use είτε c-uce). Με την διαφορά όμως ότι στο κριτήριο του Ntafos έχουμε αλυσίδες από definitions σε uses.

Πιο κάτω παραθέτω δύο screenshots όπου φαίνονται τα μονοπάτια που παραχθήκαν με το κριτήριο ALL-DU Paths και τα μονοπάτια που παραχθήκαν με το κριτήριο του Ntafos για 1-dr, για το ίδιο πρόγραμμα.



Index	Reason	Variable	Path to Cover (Nodes Sequ...
0	DPU	x	[1, 2, 3]
1	DCU	x	[1, 2, 3]
2	DCU	x	[1, 2, 3, 5]
3	DPU	x	[1, 2, 4]
4	DCU	x	[1, 2, 4]
5	DCU	x	[1, 2, 4, 5]
6	DPU	y	[1, 2, 3]
7	DCU	y	[1, 2, 3]
8	DCU	y	[1, 2, 3, 5]
9	DPU	y	[1, 2, 4]
10	DCU	y	[1, 2, 4]
11	DCU	y	[1, 2, 4, 5]

Σχήμα 8.13: Μονοπάτια που παραχθήκαν με το κριτήριο ALL-DU Paths



Index	Reason-Last use	Last Variable	Path to Cover (Nodes Sequen...
0	1-dr interaction - predicate use	x	[1, 2, 3]
1	1-dr interaction - computational use	x	[1, 2, 3]
2	1-dr interaction - computational use	x	[1, 2, 3, 5]
3	1-dr interaction - predicate use	x	[1, 2, 4]
4	1-dr interaction - computational use	x	[1, 2, 4]
5	1-dr interaction - computational use	x	[1, 2, 4, 5]
6	1-dr interaction - predicate use	y	[1, 2, 3]
7	1-dr interaction - computational use	y	[1, 2, 3]
8	1-dr interaction - computational use	y	[1, 2, 3, 5]
9	1-dr interaction - predicate use	y	[1, 2, 4]
10	1-dr interaction - computational use	y	[1, 2, 4]
11	1-dr interaction - computational use	y	[1, 2, 4, 5]

Σχήμα 8.14: Μονοπάτια που παραχθήκαν με το κριτήριο του Ntafos

Όπως βλέπουμε τα μονοπάτια αυτά είναι τα ίδια παρόλο που είναι διαφορετικό το κριτήριο. Επομένως τα μονοπάτια που παράγονται με το κριτήριο ALL-DU Paths εμπεριέχονται στο κριτήριο του Ntafos.

8.4. Αποτελέσματα και συμπεράσματα

Σ' αυτή την παράγραφο θα σχολιαστούν και θα παρουσιαστούν κάποια αποτελέσματα και συμπεράσματα. Για να παρθούν τα αποτελέσματα αυτά εκτελέστηκε μια σειρά δοκιμών σε ένα σύνολο δειγμάτων (προγράμματα με διαφορετικές γραμμές κώδικα και πολυπλοκότητα) που κυμαίνονται από 10 έως 200 γραμμές κώδικα. Οι δοκιμές εκτελέστηκαν σε ένα Ηλεκτρονικό Υπολογιστή με επεξεργαστή Intel Pentium 4 στα 3.6 GHz με 2 GB μνήμη Ram και JDK 1.6 και στον οποίο ήταν εγκατεστημένο λειτουργικό σύστημα Windows XP Professional (SP2). Τα προγράμματα που χρησιμοποιήθηκαν για τις δοκιμές παράχθηκαν τυχαία και δεν εξυπηρετούν κάποιο συγκεκριμένο σκοπό. Στον επόμενο πίνακα παρουσιάζονται τα αποτελέσματα για κάποια προγράμματα.

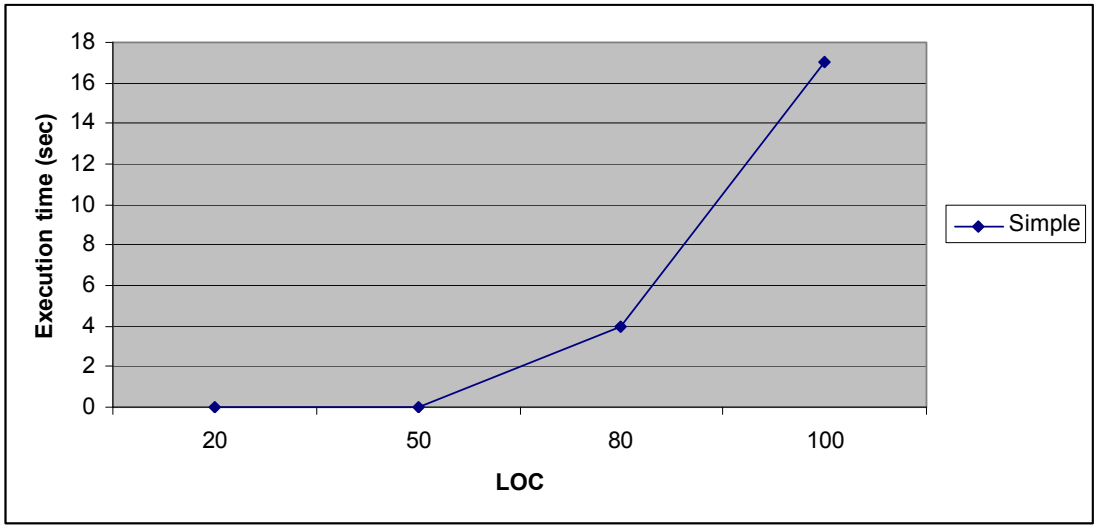
Id	LOC	# nested if	# if	Complexity	#test cases (useful chromos)	Coverage %	Evolutions	Time (sec)
1	20	0	1	Simple	8	100 %	7	0
2	20	1	2	Medium	8	100 %	16	0
3	20	2	3	High	7	85 %	304	53
4	50	0	1	Simple	10	100 %	69	0
5	50	1	2	Medium	8	100 %	42	1
6	50	2	3	High	8	85 %	304	60
7	80	0	2	Simple	14	100 %	70	4
8	80	1	2	Medium	8	100 %	58	2
9	80	2	3	High	6	85%	303	60
10	100	0	4	Simple	14	100 %	50	17
11	100	1	3	Medium	13	100 %	58	5
12	100	2	4	High	14	85%	314	62

Πίνακας 8.1: Μετρήσεις και αποτελέσματα

Σημείωση: *Η πολυπλοκότητα ενός προγράμματος χαρακτηρίζεται ως απλή (simple) αν τα if δεν περιέχουν nested ifs. Αν κάποιο if περιέχει ένα nested if, τότε η πολυπλοκότητα του χαρακτηρίζεται ως μεσαία (medium) και αν κάποιο if περιέχει δύο ή περισσότερα nested ifs τότε η πολυπλοκότητα του χαρακτηρίζεται ως ψηλή (high).*

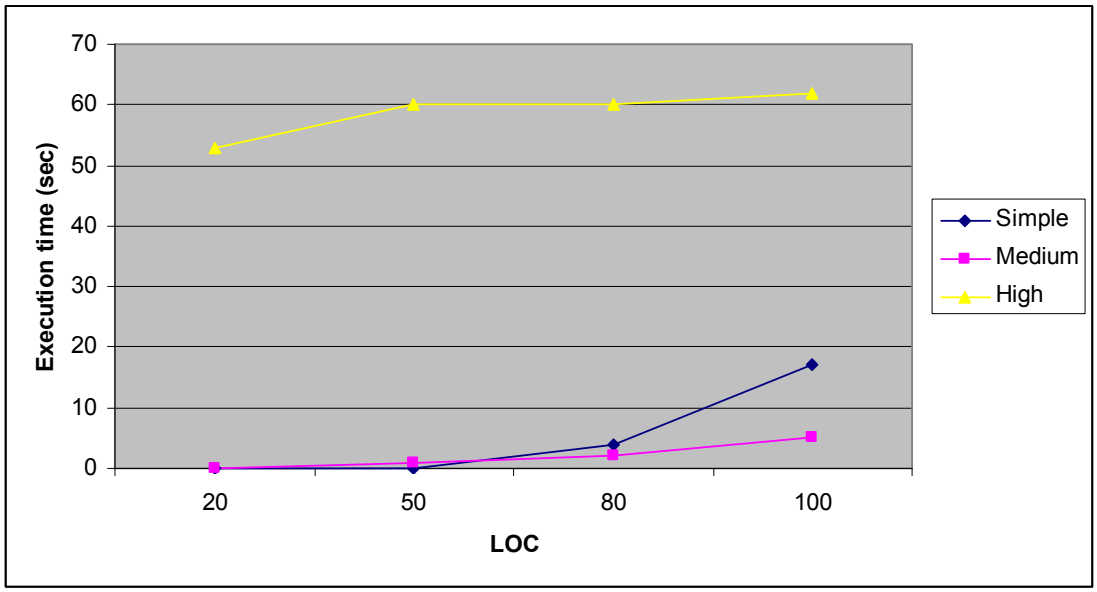
Οι συγκριτικές μετρήσεις επίδοσης αποκάλυψαν ότι για μικρά και απλά προγράμματα, ανεξαρτήτως πολυπλοκότητας, το σύστημα μας μπορεί μέσα σε πολύ λίγο χρόνο και μετά από εξέταση λίγων χρωματοσωμάτων να καλύψει τα αναγκαία μονοπάτια και να αποφανθεί για την ορθότητα του προγράμματος.

Πιο κάτω παρουσιάζεται η γραφική του χρόνου ως προς τις γραμμές κώδικα για απλά (simple) προγράμματα. Βλέπουμε ότι υπάρχει μια γραμμικότητα στην αύξηση του χρόνου σε σχέση με τις γραμμές κώδικα. Αυτό συμβαίνει και για τα προγράμματα μεσαίας (medium) πολυπλοκότητας, καθώς και για τα προγράμματα υψηλής (high) πολυπλοκότητας (δες γραφική 8.2). Πιο κάτω παρουσιάζω την γραφική του χρόνου ως προς τις γραμμές κώδικα για απλά (simple) προγράμματα, ως ενδεικτική.



Γραφική 8.1.: Οι γραμμές κώδικα σε σχέση με τον χρόνο για απλά προγράμματα

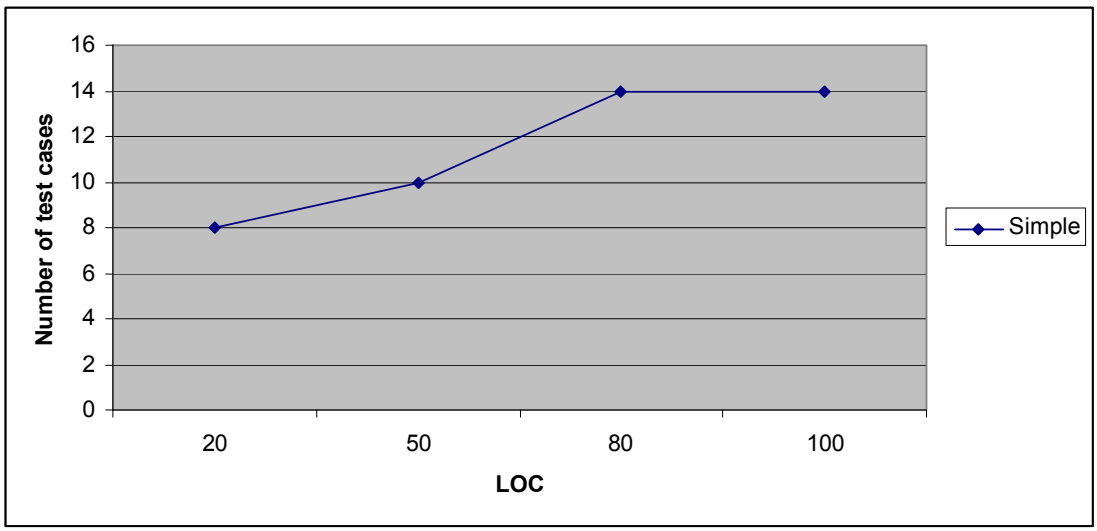
Πιο κάτω φαίνεται η γραφική του χρόνου ως προς τις γραμμές κώδικα για προγράμματα απλής, μεσαίας και υψηλής πολυπλοκότητας.



Γραφική 8.2.: Οι γραμμές κώδικα σε σχέση με τον χρόνο

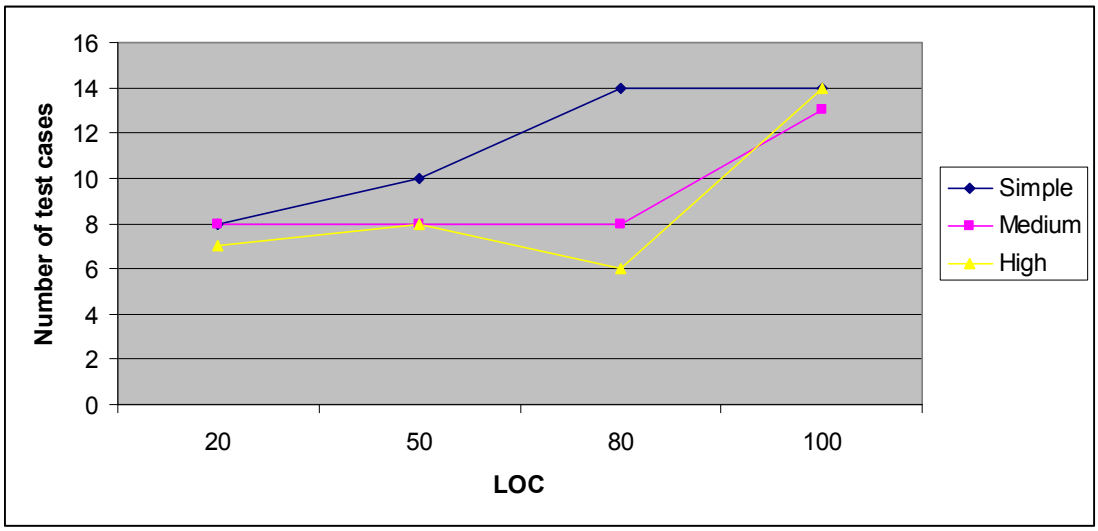
Πιο κάτω φαίνεται η γραφική ως προς τις γραμμές κώδικα σε σχέση με των αριθμών των test cases. Φαίνεται ότι υπάρχει μια αύξηση και αυτό είναι λογικό γιατί όσο μεγαλύτερο αριθμό γραμμών έχουμε σε ένα πρόγραμμα, τόσα περισσότερα μονοπάτια θα έχουμε. Άρα απαιτούνται περισσότερα test cases για κάλυψη αυτών των μονοπατιών. Η πιο κάτω γραφική είναι απλά ενδεικτική και είναι για απλά (simple) προγράμματα. Το ίδιο

συμβαίνει και με προγράμματα μεσαίας (medium) πολυπλοκότητας, καθώς και με προγράμματα υψηλής (high) πολυπλοκότητας (δες γραφική 8.4).



Γραφική 8.3.: Οι γραμμές κώδικα σε σχέση με τον αριθμό των απαιτούμενων test cases για απλά προγράμματα

Πιο κάτω φαίνεται η γραφική των γραμμών κώδικα σε σχέση με των αριθμών των test cases για προγράμματα απλής, μεσαίας και υψηλής πολυπλοκότητας.



Γραφική 8.4.: Οι γραμμές κώδικα σε σχέση με τον αριθμό των απαιτούμενων test cases

Κάτι άλλο που μπορεί να παρατηρηθεί από τον πίνακα 8.1. είναι ότι για προγράμματα με υψηλή πολυπλοκότητα είναι δύσκολο να πετύχουμε 100% κάλυψη. Αυτό που ευθύνεται είναι ότι ίσως υπάρχουν dead code επειδή τα προγράμματα που

βρίσκονται υπο έλεγχο παράχθηκαν τυχαία και δεν εξυπηρετούν κανένα σκοπό. Ή λόγω της υψηλής τους πολυπλοκότητας να μην βρέθηκαν τα κατάλληλα test case.

Τέλος πρέπει να σημειωθεί ότι οι πιο πάνω μετρήσεις και τα πιο πάνω αποτελέσματα βασίζονται σε μια ευριστική (heuristic) μέθοδο. Τα πειράματα δοκιμάστηκαν για ένα αριθμό φορών και έπειτα σημειώθηκε ο μέσος όρος τους.

8.5. Μελλοντική εργασία

Λόγω του ότι η έρευνα δεν σταματά ποτέ υπάρχουν πολλά πράγματα ακόμη που μπορούν να υλοποιηθούν ή και να βελτιστοποιηθούν.

Ένα πράγμα που μπορεί να γίνει στο μέλλον είναι να υλοποιηθούν και κάποια από τα υπόλοιπα κριτήρια των Rapps και Weyuker ή τα κριτήρια των Laski and Korel. Αυτή η εργασία μπορεί να υλοποιηθεί εύκολα αφού τα περισσότερα στοιχεία που απαιτούνται είναι ήδη υλοποιημένα. Αυτό που χρειάζεται είναι να αλλαχθεί ο αλγόριθμος παραγωγής των μονοπατιών που πρέπει να καλυφθούν. Επίσης, αφού γίνει αυτό μπορεί στη συνέχεια να γίνει μια σύγκριση των κριτηρίων που θα υλοποιηθούν με τα κριτήρια που είναι ήδη υλοποιημένα.

Ακόμη μπορεί να ελεγχτεί η υποστήριξη περισσοτέρων γλωσσών προγραμματισμού. Αν και η αρχιτεκτονική ανάλυσης προγραμμάτων υποστηρίζει την επιλογή γραμματικής οποιασδήποτε γλώσσας προγραμματισμού, εντούτοις δεν έχει δοκιμασθεί με άλλες γλώσσες εκτός από την γλώσσα προγραμματισμού Java.

Θα μπορούσαν επίσης να γίνουν βελτιστοποιήσεις στον κώδικα ώστε να γίνει πιο αποδοτικός. Υπάρχουν πολλοί βρόγχοι επανάληψης και άλλες δομές στις οποίες μπορούν να εφαρμοστούν διάφορες τεχνικές ώστε να γίνουν πιο αποδοτικές και να επιστρέφουν αποτελέσματα σε μικρότερο χρόνο.

Επίσης θα μπορούσαν να δοκιμαστούν νέες fitness functions για να βρεθεί η αποδοτικότερη, δηλαδή αυτή που καλύπτει τα περισσότερα μονοπάτια. Αυτές είναι μερικές μόνο από τις σκέψεις που μπορούν να υλοποιηθούν ως μελλοντικές εργασίες, μπορούν να υπάρξουν κι άλλες.

Βιβλιογραφία

[1] Sommerville, Ian [1982] (2007). "1.1.2 What is software engineering?", Software Engineering, 8th ed., Harlow, England: Pearson Education, P. 7. ISBN 0-321-31379-8.

[2] "IEEE Standard Glossary of Software Engineering Terminology," IEEE std 610.12-1990, 1990.

[3] F.L. Bauer (January 1969): Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct. 1968, Brussels: Scientific Affairs Division, NATO

[4] Σοφοκλέους Αναστάσης, Έλεγχος κώδικα λογισμικού: Αυτόματη Δημιουργία γράφου ελέγχου ροής και περιπτώσεις ελέγχου μέσω αλγορίθμων βελτιστοποίησης, Ατομική Μεταπτυχιακή Εργασία, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου 2004.

[5] Οικονομίδης Κύπρος, Αυτοματοποίηση ελέγχου λογισμικού: Δημιουργία γράφου ελέγχου ροής δεδομένων και έλεγχος λογισμικού με χρήση γενετικών αλγορίθμων. Ατομική Μεταπτυχιακή Εργασία, Τμήμα Πληροφορικής, Πανεπιστήμιο Κύπρου 2007.

[6] Glenford Myers, «The art of software testing». 174 Seiten, Wiley 1979 [ISBN 0471043281]

[7] William C. Hetzel, The Complete Guide to Software Testing (QED Publishing Group) Wellesley, Massachusetts 1988. [ISBN 0-89435-242-3]

[8] Rick D. Craig and Stefan P. Jaskiel, 2002. Systematic Software Testing. Artech House Publishers, London

[9] Oliver E. Cole, Looking under the covers to test web applications, 2001, STAR East Conference.

[10] Ιστοσελίδα: http://en.wikipedia.org/wiki/Cyclomatic_complexity

[11] Ιστοσελίδα: http://en.wikipedia.org/wiki/Static_testing

[12] Beizer, B. «Software testing techniques» - second edition, 1990. Boston, Mass.: International Thomson Computer Press.

[13] Doron A. Peled, «Software Reliability Methods». Publisher Springer-Verlag. P249-298, 2001.

[14] Lee Copeland, "A Practitioner's Guide to Software Test Design", STQE Publishing, 2004.

[15] J. W. Laski and B. Korel, "A data flow oriented program testing strategy", IEEE Trans. on Software Eng., Vol. SE-9, no 3, pp. 347 – 354, May 1983

[16] S. Rapps and E. J. Weyuker, "Data Flow analysis techniques for test data selection", in Proc. Sixth Int. Conf. Software Engineering, IEEE Comput. Soc., Tokyo, Japan. September 1982, pp 272-277

[17] S. Rapps and E. J. Weyuker, "Selecting Software test data using data flow information", IEEE Trans. on Software Eng., Vol. SE-11, no 4, pp. 367 – 375, April 1985

[18] P. G. Frankl and E. J. Weyuker, "An applicable family of data flow testing criteria", IEEE Trans. on Software Eng., Vol. 14, no 10, pp. 1483 – 1498, October 1988

[19] L. a. Clarke et al, "A formal Evaluation of Data Flow Path Selection Criteria", IEEE Trans. on Software Eng., Vol. 15, no 11, pp. 1318 – 1332, November 1989

[20] J. W. Laski and B. Korel, "A data flow oriented program testing strategy", IEEE Trans. on Software Eng., Vol. SE-9, no 3, pp. 347 – 354, May 1983

[21] S.C. Ntafos, "A comparison of some structural testing stategies", IEEE Trans. on Software Eng., Vol. 14, no 6, pp. 868 – 874, June 1988

[22] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, Hasan Ural, "Data Flow Testing as Model Checking", in Proc. 25th International Conference on Software Engineering (ICSE'03), IEEE.

[23] Goldberg, D. E., «Genetic Algorithms in Search, Optimization, and Machine Learning», 1989.

[24] Syswerda, G., «Uniform crossover in genetic algorithms». In Proceedings of the Third International Conference on Genetic Algorithms, 1989.

[25] Ιστοχώρος http://jgap.sourceforge.net/

[26] Andreou, A. and Sofokleous, A. 2004. Designing and implementing a layered architecture for dynamic and interactive program analysis. In Proceedings of IADIS International Conference, Lisbon, Portugal, March 2004, N. GUIMARÃES AND P. ISAÍAS, Eds., 387-397.

[27] Βιβλίο: Ανάπτυξη Εφαρμογών σε Προγραμματιστικό Περιβάλλον (Γ' Λυκείου Τεχνολογικής Κατεύθυνσης)

[28] Abdelaziz Khamis, Reem ahgat, Rana Abdelaziz: Automatic test data Generation using data flow information

[29] Ahmed S. Ghiduk, Mary Jean Harrold, Moheb R. Girgis (2007) Using Genetic Algorithms to Aid Test-Data Generation for Data-Flow Coverage

[30] Hyoung Seok Hong, Sung Deok Cha, Insup Lee, Oleg Sokolsky, Hasan Ural: Data Flow testing as Model Checking

[31] Anastasis A. Sofokleous, Andreas S. Andreou, Automatic, evolutionary test data generation for dynamic software testing, 2007