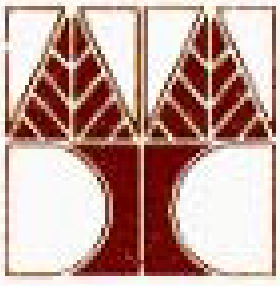
**Διατριβή**

# ΑΥΤΟΜΑΤΟΣ ΕΝΤΟΠΙΣΜΟΣ ΚΑΙ ΔΙΟΡΘΩΣΗ ΣΦΑΛΜΑΤΟΣ ΣΕ ΠΡΟΓΡΑΜΜΑΤΑ JAVA ΜΕ ΧΡΗΣΗ ΔΥΝΑΜΙΚΟΥ ΤΕΜΑΧΙΣΜΟΥ, ΕΛΕΓΧΟΥ ΜΕΤΑΛΛΑΞΗΣ ΚΑΙ ΓΕΝΕΤΙΚΩΝ ΑΛΓΟΡΙΘΜΩΝ

**Χριστιάνα Στυλιανού Σταύρου**

## ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ



## ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

**Ιανουάριος 2010**

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ**

**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ**

# Εντοπισμός και Διόρθωση Σφάλματος σε Προγράμματα Java με Χρήση Δυναμικού Τεμαχισμού, Ελέγχου Μετάλλαξης και Γενετικών Αλγορίθμων

**Χριστιάνα Στυλιανού Σταύρου**

Επιβλέπων Καθηγητής
Δρ. Ανδρέας Ανδρέου

Η Ατομική αυτή Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων του Μεταπτυχιακού Προγράμματος Προηγμένες Τεχνολογίες Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Ιανουάριος 2010

## Ευχαριστίες

Θα ήθελα να αφιερώσω την μεταπτυχιακή μου εργασία στον μικρούλι μου Ανδρέα-Ιωάννη και στον σύζυγο μου, Άριστο για την αγάπη, την υπομονή, την ενθάρρυνση σε όλη αυτή τη διαδικασία. Χωρίς την υποστήριξη και την κατανόηση του δεν θα μπορούσα να ολοκληρώσω αυτή τη δουλεία. Τις ειλικρινείς μου ευχαριστίες στον επιβλέπων καθηγητή μου, Δρ. Ανδρέα Ανδρέου για την πολύτιμη καθοδήγηση και συμβουλές κατά την εκπόνηση της παρούσας εργασίας. Ιδιαίτερες ευχαριστίες επίσης στον Δρ. Αναστάση Σοφοκλέους για την βοήθεια και όλη τη συνεργασία που είχαμε. Τέλος θα ήθελα να ευχαριστήσω την οικογένεια μου για την συνεχή υποστήριξη που μου παρέχει όλα αυτά τα χρόνια και ειδικά τη μητέρα μου γιατί είναι πάντα δίπλα μου.

## Περίληψη

Η αποσφαλμάτωση είναι μια διαδικασία περίπλοκη και χρονοβόρα, για αυτό και έχουν γίνει πολλές προσπάθειες για αυτοματοποίηση της. Έχοντας σαν στόχο την βελτίωση της αποσφαλμάτωσης, ο Τεμαχισμός Προγράμματος περιορίζει το πεδίο αναζήτησης του σφάλματος αποκλείοντας τις δηλώσεις που δεν λαμβάνουν μέρος στον υπολογισμό της λανθασμένης εξόδου. Με τον Δυναμικό Τεμαχισμό το τεμάχιο προγράμματος έχει μειωθεί σε κάποιο βαθμό, εντούτοις είναι τις περισσότερες φορές αρκετά μεγάλη.

Οι τεχνικές ελέγχου με χρήση μετάλλαξης εφαρμόζουν διάφορους τελεστές μετάλλαξης σε ένα σωστό πρόγραμμα εισάγοντας συχνά προγραμματιστικά λάθη. Δημιουργείται έτσι ένα μεγάλο σύνολο από λανθασμένα προγράμματα κάθε ένα από τα οποία περιέχει ένα συγκεκριμένο λάθος. Τα λανθασμένα προγράμματα εκτελούνται με ένα σύνολο από σενάρια ελέγχου με σκοπό τη μέτρηση της αποτελεσματικότητας τους με βάση την ικανότητα τους να εντοπίσουν τα λάθη.

Σε αυτή την εργασία προτείνεται μια καινοτόμος μεθοδολογία για εντοπισμό αλλά και διόρθωση σφάλματος σε προγράμματα Java, με χρήση Δυναμικού Τεμαχισμού, Ελέγχου Μετάλλαξης και Γενετικών Αλγορίθμων. Συγκεκριμένα, αξιοποιώντας την πληροφορία που μας δίνει ο δυναμικός τεμαχισμός προγράμματος προσπαθούμε να αλλάξουμε τη ροή του εσφαλμένου προγράμματος εφαρμόζοντας αντικαταστάσεις όχι σε ολόκληρο το πρόγραμμα αλλά μόνο στις γραμμές που περιέχονται στη τεμάχιο της λανθασμένης εκτέλεσης του προγράμματος. Για την δημιουργία των αντικαταστάσεων που θα εφαρμόσουμε σε κάθε μια από τις εν λόγω γραμμές χρησιμοποιούμε τους τελεστές μετάλλαξης του ελέγχου μετάλλαξης.

Η εύρεση της δήλωσης που περιέχει το λάθος σε συνδυασμό με την επιλογή της σωστής αντικατάστασης αποτελεί ένα πρόβλημα με πολύ μεγάλο αριθμό λύσεων. Χρησιμοποιώντας γενετικούς αλγόριθμούς αναγάγουμε το πρόβλημα αυτό σε πρόβλημα αναζήτησης.

# Περιεχόμενα

# Κεφάλαιο 1

# Εντοπισμός Σφάλματος (Fault Localization)

---



---

## 1.1 Κίνητρο

Η αύξηση της παραγωγικότητας και της ποιότητας κατά τη διαδικασία ανάπτυξης λογισμικού αποτελεί σημαντικό ερευνητικό αντικείμενο για το πεδίο της ανάπτυξης λογισμικού. Ωστόσο, για να πετύχουμε βελτίωση της παραγωγικότητας και της ποιότητας πρέπει πρώτα από όλα να βελτιώσουμε την διαδικασία ανάπτυξης λογισμικού [FFN91, HU89]. Είναι ευρέως αποδεκτό ότι τα λάθη έχουν μεγάλο αντίκτυπο στην παραγωγικότητα και την ποιότητα λογισμικού, δεν είναι λίγες οι φορές που η ύπαρξη λαθών σε ακριβά λογισμικά αποτέλεσε πρώτη είδηση στις εφημερίδες. Μερικά παραδείγματα από αυτά τα λάθη ήταν το Disney Lion King το 1994-1995, το Intel Pentium Floating-Point Division Bug το 1994, το NASA Mars Polar Lander το 1999, το Patriot Missile Defense System το 1991, το Y2K (Year 2000) Bug γύρω στο 1974, το Dangerous Viewing Ahead το 2004 [ST06]. Οι περισσότερες εταιρείες εκτιμάται ότι στην προσπάθεια τους να μειώσουν τον αριθμό των λαθών, ξοδεύουν το 50 με 80% του χρόνου ανάπτυξης σε διαδικασίες ελέγχου του λογισμικού [CW89]. Ως εκ τούτου, η μείωση της προσπάθειας ελέγχου είναι ένα σημαντικό βήμα για την αύξηση της παραγωγικότητας κατά στην διαδικασία ανάπτυξης λογισμικού.

Η διαδικασία ελέγχου απαρτίζεται από δύο υποδιαδικασίες, τον έλεγχο (testing) κατά τον οποίο εντοπίζονται τυχόν αποτυχίες (failures) που συμβαίνουν κατά την εκτέλεση του προγράμματος, και την αποσφαλμάτωση (debugging) κατά την οποία εντοπίζονται τα σφάλματα (faults) που είναι υπαίτια για τις αποτυχίες (failures) και διορθώνεται το

πρόγραμμα. Σύμφωνα με το IEEE Standard Glossary of Software Engineering Terminology, με τον όρο αποτυχία (failure) εννοούμε την αδυναμία ενός συστήματος ή συστατικού να εκτελέσει τις απαιτούμενες λειτουργίες του με βάση συγκεκριμένες απαιτήσεις απόδοσης. Ενώ σφάλμα (fault) σημαίνει ένα λάθος βήμα, διαδικασία ή ορισμός δεδομένων σε ένα πρόγραμμα. Επιπλέον για τον όρο σφάλμα (fault), χρησιμοποιούνται και άλλες συνώνυμες ονομασίες όπως: πρόβλημα (problem), error (λάθος), anomaly (ανωμαλία), ασυνέπεια (inconsistency) και bug [ST06].

Σύμφωνα με το [MY79] η διαδικασία εντοπισμού των σφαλμάτων που αποτελεί μέρος της υποδιαδικασίας αποσφαλμάτωσης καταλαμβάνει το 95% της προσπάθειας αποσφαλμάτωσης. Είναι λοιπόν προφανές ότι η ανάπτυξη μιας αποτελεσματικής διαδικασίας εντοπισμού σφάλματος είναι πολύ σημαντική.

## 1.2 Σκοπός

Σε αυτή την εργασία προτείνεται μια καινοτόμος μεθοδολογία για εντοπισμό αλλά και διόρθωση σφάλματος σε προγράμματα Java, με χρήση Δυναμικού Τεμαχισμού, Ελέγχου Μετάλλαξης και Γενετικών Αλγορίθμων. Συγκεκριμένα, αξιοποιώντας την πληροφορία που μας δίνει ο δυναμικός τεμαχισμός προγράμματος προσπαθούμε να αλλάξουμε τη ροή του εσφαλμένου προγράμματος εφαρμόζοντας αντικαταστάσεις όχι σε ολόκληρο το πρόγραμμα αλλά μόνο στις γραμμές που περιέχονται στη φέτα της λανθασμένης εκτέλεσης του προγράμματος. Για την δημιουργία των αντικαταστάσεων που θα εφαρμόσουμε σε κάθε μια από τις εν λόγω γραμμές χρησιμοποιούμε τους τελεστές μετάλλαξης του ελέγχου μετάλλαξης.

Η εύρεση της δήλωσης που περιέχει το λάθος σε συνδυασμό με την επιλογή της σωστής αντικατάστασης αποτελεί ένα πρόβλημα με πολύ μεγάλο αριθμό λύσεων. Χρησιμοποιώντας γενετικούς αλγόριθμούς αναγάγουμε το πρόβλημα αυτό σε πρόβλημα αναζήτησης

## 1.3 Ανασκόπηση Κεφαλαίων

Στα επόμενα τέσσερα κεφάλαια γίνεται μια εισαγωγή στις σχετικές τεχνολογίες. Στο Κεφάλαιο 2 γίνεται μια εκτενής αναφορά στον Τεμαχισμό Προγράμματος, ενώ στο Κεφάλαιο 3 αναφέρονται διάφοροι αλγόριθμοι που έχουν προταθεί για Δυναμικό Τεμαχισμό Προγράμματος. Στο Κεφάλαιο 4 γίνεται μια εισαγωγή στους Γενετικούς αλγορίθμους. Στο Κεφάλαιο 5 , παρουσιάζεται η τεχνική του ελέγχου με χρήση μετάλλαξης, δίνοντας ιδιαίτερη έμφαση στον έλεγχο με χρήση μετάλλαξης για αντικειμενοστραφή προγράμματα. Η μεθοδολογία που προτείνει η παρούσα εργασία περιγράφεται στο Κεφάλαιο 6. Επιπλέον σε αυτό το κεφάλαιο δίνεται μια λεπτομερής αναφορά στην αναπαράσταση του προβλήματος ως πρόβλημα γενετικών αλγορίθμων, δίνονται οι λεπτομέρειες της υλοποίησης και παρατίθενται τα αποτελέσματα που έχουν εξαχθεί μετά από πειραματική χρήση του εργαλείου. Τέλος, στο Κεφάλαιο 7 γίνεται αξιολόγηση και σύγκριση της παρούσας εργασίας με άλλες προτάσεις.

# Κεφάλαιο 2

# Εντοπισμός Σφάλματος (Fault Localization)

---

2.1 Προτάσεις / Μεθοδολογίες για Εντοπισμό Σφάλματος

---

## 2.1 Προτάσεις / Μεθοδολογίες για Εντοπισμό Σφάλματος

Οι προγραμματιστές έρχονται καθημερινά αντιμέτωποι με την ύπαρξη λαθών σε προγράμματα, όταν παρατηρούν ότι η έξοδος του προγράμματος παρεκκλίνει από την αναμενόμενη έξοδο. Μια καθιερωμένη διαδικασία αποσφαλμάτωσης αποτελείται από την εισαγωγή σημείων διακοπής (breakpoints), την επανεκτέλεση του προγράμματος με βάση την είσοδο που προκάλεσε την αποτυχία και την εξέταση της κατάστασης του προγράμματος, δηλαδή τις τιμές των διαφόρων μεταβλητών κ.τ.λ. έτσι ώστε να ανακαλύψουν το λόγο που προκάλεσε την παραγωγή λανθασμένης εξόδου. Κατά τη διάρκεια αυτής της διαδικασίας οι προγραμματιστές πρέπει να αποφασίσουν ποιο κομμάτι της εκτέλεσης πρέπει να εξετάσουν για να απομονώσουν το λάθος. Ωστόσο η διαδικασία της εξέτασης είναι συχνά βαρετή και χρονοβόρα. Αυτό ώθησε την ανάπτυξη αυτοματοποιημένων τεχνικών ελέγχου οι οποίες αξιοποιούν την επεξεργαστική δύναμη των υπολογιστών με στόχο να κάνουν τη διαδικασία αποσφαλμάτωσης λιγότερο βαρετή και χρονοβόρα.

Έχουν λοιπόν προταθεί διαφορές τεχνικές για αυτοματοποίηση της αποσφαλμάτωσης από πληθώρα ερευνητών. Μερικές από αυτές τις τεχνικές είναι η αποσφαλμάτωση delta, διάφορες παραλλαγές δυναμικού προγραμματισμού, failure inducing chops και predicate switching. Πιο κάτω παραθέτουμε μερικές από τις εργασίες που έχουν προταθεί με χρονολογική σειρά.

**Fault Localization Using Execution Slices and Dataflow Tests**

Στο [AHLW95] προτάθηκε μια ευριστική τεχνική η οποία υποστηρίζει την εκτέλεση τεμαχισμού (slicing) και κυβοποίησης (Dicing) με βάση τα σενάρια ελέγχου. Οι Agrawal, Horgan, London και Wong παρατήρησαν ότι ένα λάθος εντοπίζεται στο τεμάχιο που αντιπροσωπεύει το σενάριο ελέγχου το οποίο έχει αποτύχει κατά την εκτέλεση του προγράμματος. Για να βρούμε το λάθος δίνεται προσοχή σε αυτό το τεμάχιο και αγνοείται το υπόλοιπο πρόγραμμα. Για να ελαττώσουμε το πεδίο έρευνας για τον εντοπισμό της λανθασμένης δήλωσης, υποθέτουμε ότι η δήλωση αυτή δεν περιέχεται σε οποιονδήποτε τεμάχιο που αντιστοιχεί σε ένα επιτυχές σενάριο ελέγχου. Έτσι συγκεντρωνόμαστε στις δηλώσεις του αποτυχημένου τεμαχίου στο οποίο δεν περιέχονται στο επιτυχημένο τεμάχιο, αυτό το κομμάτι του τεμαχίου το ονομάζουμε dice. Τα αποτελέσματα των πειραμάτων που διεξήχθησαν έδειξαν ότι η μέθοδος αυτή εντοπίζει σφάλματα τα οποία εμφύτεψαν στον κώδικα ανεξάρτητοι παρατηρητές. Ωστόσο κάποια από τα λάθη τα οποία εμφύτεψαν οι παρατηρητές δεν εντοπίστηκαν και επιπλέον μερικά δεν περιέχονταν ποτέ στα dices ακόμα και στις περιπτώσεις που περιέχονταν στα τεμάχια προγράμματος. Επιπλέον παρατηρήθηκε ότι αν και είναι απίθανος ο εντοπισμός σφαλμάτων παράλειψης (omission), η τεχνική αυτή εντόπισε συγκεκριμένα λάθη παράληψης. Ωστόσο στα κατοπινά χρόνια όπως θα δούμε και παρακάτω έγιναν περισσότερο εξεζητημένες προτάσεις μεθοδολογιών εντοπισμού σφάλματος.

**Fault Localization With Nearest Neighbor Queries**

Οι Renieris και Reiss [RR03] πρότειναν μια μέθοδο για εντοπισμό του λάθους η οποία υποθέτει την ύπαρξη μιας λανθασμένης εκτέλεσης και ενός μεγαλύτερου αριθμού επιτυχών εκτελέσεων. Στη συνέχεια επιλέγει με βάση το κριτήριο απόστασης την επιτυχημένη εκτέλεση που μοιάζει περισσότερο με την λανθασμένη εκτέλεση, συγκρίνει τα στοιχεία που ανήκουν στις δύο εκτελέσεις και παράγει μια λίστα με τα «ύποπτα» κομμάτια του προγράμματος. Η προτεινόμενη μέθοδος είναι ευρέως εφαρμόσιμη μιας και δεν προϋποθέτει γνώση των τιμών εισόδου του προγράμματος και καμία επιπλέον πληροφορία από τον χρήστη εκτός από τη κατηγοριοποίηση των εκτελέσεων σε «σωστές» και «λανθασμένες». Με τον όρο κομμάτια του προγράμματος εννοούμε κομμάτια κώδικα που μπορεί να είναι βασικά blocks, μέθοδοι, μονοπάτια ή τεμάχια προγράμματος (slices). Στην ουσία σπάζουν την λανθασμένη εκτέλεση καθώς και όλες τις σωστές εκτελέσεις σε κομμάτια κώδικα. Κάθε εκτέλεση αναπαριστάται από

ένα δυαδικό πίνακα. Χρησιμοποιώντας μια συνάρτηση υπολογισμού της απόστασης βασισμένη στην απόσταση Hamming, υπολογίζεται η διαφορά μεταξύ της λανθασμένης εκτέλεσης και κάθε επιτυχούς εκτέλεσης και επιλέγεται η επιτυχή εκτέλεση που είναι πιο κοντά στη λανθασμένη εκτέλεση. Η απόσταση Hamming ορίζεται ως ο αριθμός των θέσεων που διαφωνούν οι δύο πίνακες, με τη διαφοροποίηση ότι δεν λαμβάνονται υπόψη τα χαρακτηριστικά που απουσιάζουν από τη λανθασμένη εκτέλεση, αλλά μόνο εκείνα τα χαρακτηριστικά που υπάρχουν στην λανθασμένη εκτέλεση.

**Using Program Slicing to identify Faults in Software**

Στο [BCHW05] εξετάζεται η σχέση μεταξύ των φετών ενός προγράμματος και των σφαλμάτων που υπάρχουν στο πρόγραμμα, με σκοπό να ερευνηθεί κατά πόσο τα χαρακτηριστικά των φετών ενός προγράμματος μπορούν να χρησιμοποιηθούν στον προσδιορισμό συστατικών του προγράμματος που πιθανώς να περιέχουν σφάλματα. Για τον χαρακτηρισμό της κατανομής τεμαχισμού ενός συστατικού προγράμματος χρησιμοποιούνται μετρικές τεμαχισμού και κλάσεις εξαρτήσεων (dependence clusters) και στη συνέχεια αναλύονται τα σφάλματα στο συστατικό. Η ύπαρξη λαθών σε ένα πρόγραμμα αυξάνει την πιθανότητα το πρόγραμμα να καταστεί ασταθές και να επιφέρει προβλήματα κατά την ανάπτυξη και την εξέλιξη του προγράμματος. Ο εντοπισμός εκείνων των συστατικών του προγράμματος που πιθανώς να περιέχουν το σφάλμα είναι μια δύσκολη διαδικασία. Ερευνάται λοιπόν κατά πόσο η χρήση τεμαχισμού προγράμματος μπορεί να επεκταθεί σαν ένα αξιόπιστο εργαλείο για την πρόβλεψη συστατικών του λογισμικού που είναι επιρρεπή σε σφάλματα. Η έρευνα έχει διεξαχθεί σε δύο επίπεδα, στο πρώτο επίπεδο έχουν διατυπωθεί οι πιο κάτω υποθέσεις αναφορικά με αυτή τη σχέση: (1) Τα μικρά προγράμματα έχουν λιγότερα σφάλματα. (2) Ο κώδικας που συμπεριλαμβάνεται σε πολλά τεμάχια προγράμματος περιέχει λιγότερα σφάλματα. (3) Τα σφάλματα έχουν περισσότερες πιθανότητες να συμβούν σε κώδικα που είναι αλληλοεξαρτώμενος σε μεγάλο βαθμό. Στη συνέχεια εξετάζεται το ποσοστό του κώδικα στα τεμάχια προγράμματος που περιέχεται στο dependence cluster του προγράμματος. Στο δεύτερο επίπεδο της εν λόγω εργασίας έχουν ελεγχθεί οι πιο πάνω υποθέσεις σε ένα open source σύστημα, το οποίο ονομάζεται Barcode και έχουν καταλήξει στο συμπέρασμα ότι ισχύουν.

**Experimental evaluation of using dynamic slices for fault location**

Στο [ZHGG05] οι Zhang, He και Gupta υλοποίησαν ένα πλαίσιο εργασίας (framework) δυναμικού τεμαχισμού το οποίο επιτρέπει τον τεμαχισμό προγραμμάτων με μεγάλο μονοπάτι εκτέλεσης. Το πλαίσιο εργασίας χρησιμοποιήθηκε για την υλοποίηση τριών παραλλαγών αλγορίθμων δυναμικού τεμαχισμού: Τεμαχισμός Δεδομένων (Data Slicing), Πλήρης Τεμαχισμός (Full Slicing), Σχετικός Τεμαχισμός (Relevant Slicing). Κατά τον τεμαχισμό Δεδομένων περιλαμβάνονται στα τεμάχια προγράμματος οι δηλώσεις που επηρεάζουν άμεσα ή έμμεσα μέσω αλυσιδωτών εξαρτήσεων δεδομένων τον υπολογισμό της λανθασμένης τιμής εξό δ υ Κατά τον Πλήρη τεμαχισμό περιλαμβάνονται στο τεμάχιο προγράμματος οι δηλώσεις που επηρεάζουν άμεσα ή έμμεσα τον υπολογισμό της λανθασμένης τιμής εξόδου μέσω αλυσιδωτών δυναμικών δεδομένων ή/και εξαρτήσεων ελέγχου. Ο Σχετικός Τεμαχισμός λαμβάνει επίσης υπόψη τις εξαρτήσεις δεδομένων και ελέγχου αλλά επιπρόσθετα περιλαμβάνει και κατηγορήματα τα οποία αν και στην ουσία δεν επηρέασαν την έξοδο θα μπορούσαν να την είχαν επηρεάσει αν είχαν αποτιμηθεί διαφορετικά, καθώς και τις άμεσες εξαρτήσεις δεδομένων για αυτά τα κατηγορήματα και τις αλυσιδωτές εξαρτήσεις δεδομένων και ελέγχου σε αυτές. Τα πειράματα που πραγματοποίησαν έδειξαν ότι τα τεμάχια Δεδομένων είναι μικρότερες αλλά περιλαμβάνουν κατά μέσο όρο λιγότερες λανθασμένες δηλώσεις. Τα Πλήρης τεμάχια είναι μεγαλύτερα σε μέγεθος από τα τεμάχια Δεδομένων και περιλαμβάνουν μεγαλύτερο αριθμό λανθασμένων δηλώσεων σε σχέση με τα τεμάχια Δεδομένων. Τα Σχετικά τεμάχια λίγο πιο μεγάλα από ότι τα Πλήρη τεμάχια αλλά περιλαμβάνουν τις περισσότερες λανθασμένες δηλώσεις. Όσον αφορά την επίδοση (performance) ο αλγόριθμος τεμαχισμού Δεδομένων είναι ο ταχύτερος ενώ ο Σχετικός αλγόριθμος τεμαχισμού είναι ο πιο αργός.

**Locating Faulty Code Using Failure - Inducing Chops**

Στο [GHZG05] προτάθηκε μια καινούργια προσέγγιση η οποία συνδυάζει το δυναμικό του αλγορίθμου αποσφαλμάτωσης delta, με το όφελος του προς τα εμπρός και προς τα πίσω τεμαχισμού προγράμματος για να περιορίσει ακόμα περισσότερο την αναζήτηση του λάθος σε ένα τεμάχιο προγράμματος.

Σε αυτό το σημείο θα δώσο ψε ένα σύντομο ορισμό της αποσφαλμάτωσης delta. Αποσφαλμάτωση delta λοιπόν είναι η διαδικασία προσδιορισμού των αιτιών που καθορίζουν τη συμπεριφορά ενός προγράμματος εστιάζοντας στις διαφορές (deltas)

μεταξύ της τελευταίας και της προηγούμενης έκδοσης του προγράμματος. Στη συνέχεια η προσέγγιση αποσφαλμάτωσης delta εφαρμόζεται για την απλοποίηση και την απομόνωση της εισόδου που προκαλεί την αποτυχία του προγράμματος Για την απλοποίηση ενός ανεπιτυχούς σεναρίου ελέγχου, ο αλγόριθμος αποσφαλμάτωσης delta βρίσκει το μικρότερο σενάριο ελέγχου για το οποίο η αφαίρεση μιας και μόνο οντότητας εισόδου, απαλείφει την αποτυχία.

Ας δούμε όμως αναλυτικότερα την πρόταση των Gupta , Zhang, He και Gupta [GHZG05]. Κατά πρώτο χρησιμοποιείται η αποσφαλμάτωση delta για την εύρεση μιας απλοποιημένης εισόδου που να προκαλεί αποτυχία του προγράμματος (failure) ή για την απομόνωση της μικρότερης διαφοράς στην είσοδο που να προκαλεί αποτυχία του προγράμματος. Στη συνέχεια υπολογίζεται η τομή των δηλώσεων του προς τα εμπρός τεμαχίου με βάση την είσοδο που προκαλεί αποτυχία του προγράμματος και των δηλώσεων που περιέχονται στο προς τα πίσω τεμάχιο προγράμματος με βάση τη λανθασμένη τιμή εξόδου, έτσι ώστε να εντοπιστεί ο κώδικας που πιθανώς να περιέχει το λάθος. Αναφερόμαστε στις δηλώσεις που περιέχονται στην τομή του προς τα εμπρός και προς τα πίσω τεμαχίου με την ορολογία ‘το τεμάχιο που προκαλεί την αποτυχία’ (failure inducing chop). Οι failure inducing chops αναμένεται να είναι πιο μικρές σε μέγεθος από τα προς τα πίσω τεμάχια μιας και περιέχουν μόνο αυτές τις δηλώσεις των δυναμικών τεμαχλιων που επηρεάζονται από τη μικρότερη είσοδο που προκαλεί αποτυχία στο πρόγραμμα. Η τεχνική αυτή έχει υλοποιηθεί και έχει δοκιμαστεί με λανθασμένες εκδοχές διαφόρων προγραμμάτων για να αξιολογηθεί. Τα αποτελέσματα δείχνουν ότι κατά μέσο όρο η λανθασμένη δήλωση περιέχεται στο failure inducing chop στο 43 με 100% των περιπτώσεων. Επιπλέον ο μέσος όρος του μεγέθους του failure inducing chop είναι 64 με 73% του μεγέθους του προς τα πίσω τεμαχίου προγράμματος και μόνο 7 με 14% του μεγέθους ολόκληρου του προγράμματος. Είναι λοιπόν εμφανές ότι οι failure inducing chops μειώνουν κατά πολύ το μέγεθος του χώρου αναζήτησης σε σύγκριση με τα δυναμικά τεμάχια, χωρίς κανένα σημαντικό συμβιβασμό στην ικανότητα τους να εντοπίζουν τον λανθασμένο κώδικα. Τέλος η τεχνική έχει εφαρμοστεί σε αρκετά προγράμματα με γνωστά λάθη που σχετίζονται με προβλήματα μνήμης π.χ. buffer overflow. Το failure inducing chop σε αρκετές από αυτές τις περιπτώσεις περιέχει 2 με 4 δηλώσεις, μεταξύ αυτών και τη δήλωση που προκαλεί το σφάλμα μνήμης.

**The Efficiency of Critical Slicing in Fault Localization**

Η τεχνική του κρίσιμου τεμαχισμού προγράμματος είναι στην ουσία ο δυναμικός προγραμματισμός εφαρμοσμένος στην μεθοδολογία ελέγχου μετάλλαξης. Στο [KWRK05] εξετάζεται η αποτελεσματικότητα της τεχνικής κρίσιμου τεμαχισμού (critical slicing) η οποία επιτυγχάνεται με τη χρήση του εργαλείου Mothra για έλεγχο μετάλλαξης (mutation testing). Η επίδοση του Mothra παρουσιάζεται κατά πρώτο σε σχέση με την μείωση στα μεγέθη των κρίσιμων φετών και κατά δεύτερο σε σχέση με τον αριθμό των κρίσιμων φετών που περιέχουν τις δηλώσεις σφάλματος. Τα πειράματα που έχουν διεξαχθεί στο πλαίσιο της εν λόγο εργασίας έχουν δείξει ότι ο κρίσιμος τεμαχισμός είναι αποτελεσματικός για τον εντοπισμό των εσφαλμένων δηλώσεων. Επίσης το εργαλείο ελέγχου μετάλλαξης Mothra είναι χρήσιμο για την διεξαγωγή των πειραμάτων κρίσιμου τεμαχισμού για τον καθορισμό της συμπεριφοράς του προγράμματος. Το σύστημα υποβοηθά την διεξαγωγή πειραμάτων με την αλληλεπίδραση του ατόμου που ασκεί τον έλεγχο, το άτομο αυτό μπορεί να επιλέξει ένα ή περισσότερους τελεστές μετάλλαξης και να δει τα αποτελέσματα, δηλαδή τα κρίσιμα τεμάχια και τις εσφαλμένες δηλώσεις που έχουν εντοπιστεί. Ο κρίσιμος τεμαχισμός είναι μια αποτελεσματική μέθοδος για τον εντοπισμό και την αναγνώριση των σφαλμάτων στον κώδικα. Αυτή η τεχνική σε συνδυασμό με το εργαλείο Mothra είναι μια προσέγγιση χαμηλού κόστους που επιτυγχάνει μεγάλο ποσοστό θετικής κάλυψης της ανάλυσης καθώς και μια σημαντική μείωση στα κρίσιμα τεμάχια. Η αναλογία όλων των κρίσιμων τεμαχίων σε σχέση με το αρχικό πρόγραμμα κυμαίνεται από 0.027 μέχρι 0.50, με μέση μείωση ίση με 0.23. Ωστόσο η περιεκτική ανάλυση δείχνει ότι το μέσο ποσοστό των κρίσιμων φετών 59.33% μπορεί να μειωθεί με τεχνικές κρίσιμου τεμαχισμού.

**Locating Faults through Automated Predicate Switching**

Στο [ZGG06b] οι Zhang και Gupta πρότειναν μια τεχνική η οποία βασίζεται στην παρατήρηση ότι ένα λανθασμένο πρόγραμμα μπορεί να τερματίσει επιτυχώς αν αλλάξουμε το αποτέλεσμα ενός κατηγορήματος κατά την εκτέλεση του προγράμματος μιας και θα αλλάξει και η ροή ελέγχου του προγράμματος. Εξετάζοντας μάλιστα το αλλαγμένο κατηγόρημα, το οποίο ονόμασαν κρίσιμο κατηγόρημα (critical predicate)

μπορούμε να αναγνωρίσουμε την αιτία του λάθος στο πρόγραμμα. Στη συνέχεια παρουσιάστηκε ένας αλγόριθμος για εντοπισμό critical predicates. Και προτάθηκε η συσχέτιση των critical predicates με τον δυναμικό τεμαχισμό έτσι ώστε να μειωθεί η προσπάθεια που καταβάλλεται για εντοπισμό του σφάλματος. Συγκεκριμένα αφού εντοπιστεί το critical predicate παράγεται το bidirectional chop (BiChop) το οποίο ορίζεται ως η τομή του bidirectional slice με το failure inducing chop που έχει παρουσιαστεί σε προηγούμενη δουλειά [GHZG05].

**Pruning dynamic slices with confidence**

Οι Zhang και Gupta πρότειναν μια στρατηγική 'κλαδέματος' του δυναμικού τεμαχίου για τον προσδιορισμός ενός υποσυνόλου δηλώσεων που πιθανώς να είναι υπεύθυνες για την παραγωγή της λανθασμένης τιμής [ZGG06c]. Η στρατηγική τους στηρίχθηκε στην παρατήρηση ότι μερικές από τις δηλώσεις που συμμετείχαν στον υπολογισμό της λανθασμένης τιμής μπορεί επίσης να συμμετείχαν στον υπολογισμό ορθών τιμών. Για κάθε τέτοια δήλωση που περιέχεται στο δυναμικό τεμάχιο, υπολογίζουμε μια τιμή 'σιγουριάς' (confidence value) με εύρος τιμών μεταξύ μηδέν και ένα με βάση την κατανομή των τιμών της εκτελέσιμης δήλωσης. Όσο μεγαλύτερη είναι η τιμή 'σιγουριάς' τόσο μεγαλύτερη είναι και η πιθανότητα η εκτέλεση της δήλωσης να παράγει ορθή τιμή. Για μια λανθασμένη εκτέλεση η οποία περιέχει ένα μόνο λάθος, απόδειξαν ότι το 'κλάδεμα' του δυναμικού τεμαχίου αποκλείοντας μόνο τις δηλώσεις με τιμή 'σιγουριάς' ίσον με ένα μειώνει κατά πολύ το μέγεθος του δυναμικού τεμαχίου, ενώ παράλληλα διατηρεί την λανθασμένη δήλωση στο τεμάχιο. Τα πειράματα που διεξήχθησαν απέδειξαν ότι ο αριθμός των ξεχωριστών δηλώσεων στο 'κλαδεμένο' τεμάχιο προγράμματος είναι 1.79 με 26.93 φορές μικρότερος από τον αριθμό των δηλώσεων σε ολόκληρο τη τεμάχιο. Επιπλέον η τιμή 'σιγουριάς' αναθέτει προτεραιότητες στις δηλώσεις του τεμαχίου προγράμματος σύμφωνα με την πιθανότητα που έχο υν να είναι λανθασμένες. Έχει αποδειχθεί ότι αν εξεταστούν οι δηλώσεις αρχίζοντας από αυτή με τη μικρότερη τιμή 'σιγουριάς' μειώνεται η προσπάθεια εντοπισμού του σφάλματος.

**Towards Locating Execution Omission Errors**

Στο [ZTGG07] προτάθηκε μια τεχνική για εντοπισμό μιας ξεχωριστής κατηγορίας σφαλμάτων, των execution omission errors. Αυτά τα λάθη οφείλονται στην παράλειψη

εκτέλεσης μέρους του προγράμματος. Οι παραδοσιακές τεχνικές δυναμικής ανάλυσης επικεντρώνονται στο τι συμβαίνει κατά την εκτέλεση του προγράμματος, για αυτό και αδυνατούν να εντοπίσουν execution omission σφάλματα μιας και αυτά σχετίζονται με το τι δεν συνέβη κατά την εκτέλεση του προγράμματος. Οι έμμεσες εξαρτήσεις (implicit dependences) είναι εξαρτήσεις που φυσιολογικά είναι μη ορατές λόγο της μη-εκτέλεσης ενός αριθμού δηλώσεων. Στο [ZTGG07] προτάθηκε μια καινούργια δυναμική μέθοδος η οποία καθιστά δυνατό τον εντοπισμό έμμεσων εξαρτήσεων (implicit dependences), οι οποίες παρουσιάζονται κατά την επανακτέλεση του προγράμματος αλλάζοντας το στιγμιότυπο συγκεκριμένου κατηγορήματος και ευθυγραμμίζοντας την αρχική και την αλλαγμένη εκτέλεση του προγράμματος. Επιπλέον προτείνεται μια διαδικασία καθοδηγούμενη από τις απαιτήσεις η οποία εκμεταλλεύεται την ανάλυση αυτοπεποίθησης (confidence analysis) για τον καθορισμό του μικρότερου 'κλαδεμένου' τεμαχίου (minimal pruned slice) και στη συνέχεια αναγνωρίζει έμμεσες εξαρτήσεις αρχίζοντας από το ελάχιστο τεμάχιο αποφεύγοντας την επικύρωση μεγάλου αριθμού πιθανών εξαρτήσεων. Τα αποτελέσματα των πειραμάτων που έγιναν έδειξαν ότι τα execution omission σφάλματα μπορούν εύκολα να εντοπιστούν με χρήση της προτεινόμενης μεθόδου και μόνο λίγες ακμές έμμεσων εξαρτήσεων χρειάζεται να αναγνωριστούν.

**A Study of Effectiveness of Dynamic Slicing in Locating Real Faults**

Οι Zhang και Gupta στο [ZGG07b] παρουσίασαν μια πειραματική αξιολόγηση της αποτελεσματικότητας των δυναμικών φετών προγράμματος για τον εντοπισμό εσφαλμένων δηλώσεων σε ένα πρόγραμμα. Συγκεκριμένα χρησιμοποίησαν τρία είδη αλγορίθμων δυναμικού τεμαχισμού: Τεμαχισμός Δεδομένων (Data Slicing), Πλήρης Τεμαχισμός (Full Slicing), Σχετικός Τεμαχισμός (Relevant Slicing). Σημειώνεται ότι έχει ήδη δοθεί ένας σύντομος ορισμός για αυτούς τους τύπους δυναμικών φετών στα πλαίσια της περιγραφής της εργασίας [ZHGG05] - Experimental evaluation of using dynamic slices for fault location. Η παρούσα εργασία [ZGG07b] αποτελεί την πρώτη εργασία για την οποία έχουν χρησιμοποιηθεί πραγματικά λάθη τα οποία έχουν αναφερθεί από χρήστες εφαρμογών που χρησιμοποιούνται ευρέως. Τα πειράματα που έγιναν στις εν λόγο εφαρμογές έδειξαν ότι τα τεμάχια Δεδομένων (Data Slices) είναι αποτελεσματικά για σφάλματα που σχετίζονται με προβλήματα μνήμης. Για τα υπόλοιπα σφάλματα ο Πλήρης Τεμαχισμός (Full slicing) ήταν επαρκή. Ενώ ο Σχετικός

Τεμαχισμός (Relevant slicing) δεν χρειάστηκε να χρησιμοποιηθεί για κανένα είδος λάθους. Επιπλέον εξάχθηκε το συμπέρασμα ότι ακόμα και αν το μέγεθος του τεμαχίου είναι μεγάλο μπορεί να χρειαστεί να εξεταστεί μόνο ένα υποσύνολο των δηλώσεων που περιέχονται στο τεμάχιο πριν να εντοπιστεί η λανθασμένη δήλωση.

**Locating Faulty Code By Multiple Points Slicing**

Στο [ZGG07] παρατήρησαν ότι σε κάθε διαφορετικό τύπο δυναμικού τεμαχισμού υπάρχουν ξεχωριστές περιπτώσεις για τις οποίες δεν είναι εφαρμόσιμος αυτός ο τύπος τεμαχισμού. Προτείνεται λοιπόν η υποστήριξη πολλαπλών τύπων δυναμικού τεμαχισμού για να επιτυγχάνεται ο χειρισμός ενός μεγαλύτερου εύρους περιπτώσεων. Επιπλέον με την υποστήριξη πολλαπλών τύπων δυναμικού τεμαχισμού δίνεται η δυνατότητα υπολογισμού ενός δυναμικού τεμαχίου πολλαπλών σημείων (multiple points dynamic slice) η οποία αποτελεί την τομή των διαφορετικών τύπων φετών. Στη συγκεκριμένη εργασία έγινε χρήση τριών διαφορετικών τύπων αλγορίθμων δυναμικού τεμαχισμού: δυναμικός τεμαχισμός προς τα εμπρός (Forward Dynamic Slicing), προς τα πίσω (Backward Dynamic Slicing) και προς τις δύο κατευθύνσεις (Bidirectional Dynamic Slicing) και έχει αποδειχτεί ότι τα δυναμικά τεμάχια πολλαπλών σημείων έχουν σημαντικά πιο μικρό μέγεθος από οποιοδήποτε τεμαχίου από αυτές των τριών τύπων δυναμικού τεμαχισμού.

**Software Fault Localization Based on Testing Requirements and Program Slice**

Μια ευρηστική προσέγγιση για εντοπισμό του σφάλματος με βάση προτεραιότητες προτείνεται στο [JJF07]. Για ένα πρόγραμμα $P$, όπου $T$ είναι το σύνολο όλων των σεναρίων ελέγχου, $Req(t_i)$ το σύνολο των απαιτήσεων που ικανοποιεί κάθε σενάριο ελέγχου $t_i$, $w_t$ ένα σενάριο ελέγχου κατά το οποίο έχουμε εσφαλμένη έξοδο, χρειάζεται να εντοπίσουμε το λάθος. Επιλέγονται τρία επιτυχή σενάρια ελέγχου και ένα ανεπιτυχές σενάριο ελέγχου σύμφωνα με τον αριθμό στοιχείων του $Req(w_t, t_i)$. Στη συνέχεια γίνεται συνδυασμός εμπειρικών παρατηρήσεων, της ευριστικής μεθόδου και της τεχνικής τεμαχισμού προγράμματος για να δοθούν προτεραιότητες στα μέρη του κώδικα. Οι προτεραιότητες δίνονται με βάση τη λογική ότι ένα κομμάτι κώδικα όσο πιο πολλές φορές περιέχεται στην εκτέλεση ενός επιτυχούς σεναρίου τόσο λιγότερη είναι η πιθανότητα να περιέχει το λάθος και αντίθετα. Έτσι τα μέρη του κώδικα που θεωρούνται περισσότερο ‘ύποπτα’ εξετάζονται πρώτα και ο χώρος αναζήτησης

αυξάνεται σταδιακά προσθέτοντας περισσότερα μέρη κώδικα με βάση τη προτεραιότητα. Η προσέγγιση αυτή έχει αξιολογηθεί πειραματικά και έχουν εξαχθεί τα πιο κάτω συμπεράσματα: στο 80.5% των περιπτώσεων, το λάθος εντοπίζεται στα αρχικά στάδια του αλγορίθμου πριν ή κατά το στάδιο εκλέπτυνσης. Στη χειρότερη περίπτωση θα εξεταστεί ολόκληρο το δυναμικό τεμάχιο για ένα δεδομένο σενάριο ελέγχου wt, ωστόσο οι πιθανότητες για αυτό το σενάριο είναι μόνο 0.7%, που σημαίνει ότι σπάνια συμβαίνει αυτό το σενάριο.

Στην παρούσα εργασία προτείνεται μια καινοτόμος τεχνική η οποία ασχολείται με τον **εντοπισμό σφάλματος** σε **Java προγράμματα**. Εκτός όμως από τον εντοπισμό του σφάλματος σκοπό έχει να προτείνει μια πιθανή **αντικατάσταση για διόρθωση του εν λόγω σφάλματος**. Η τεχνική αυτή συνδυάζει τις τεχνολογίες του **δυναμικού προγραμματισμού**, **γενετικών αλγορίθμων** και του **ελέγχου με χρήση αντικατάστασης (mutation testing)**.

## Κεφάλαιο 3

# Τεμαχισμός Προγράμματος (Program Slicing)

---



---

## 3.1 Εισαγωγή

Πολλές από τις φάσεις του Software Engineering, όπως η αποσφαλμάτωση (debugging), ο έλεγχος (testing), το re-engineering, η κατανόηση του προγράμματος (program comprehension) και software measurement, γίνονται ευκολότερα αν σπάσουμε το πρόγραμμα μας σε μικρότερα κομμάτια έτσι που να μπορούμε να επικεντρωθούμε στο κομμάτι που μας ενδιαφέρει.

Διάφορες τεχνικές που είχαν προταθεί στο παρελθόν όπως το Information Hiding ([PAR72]), το Data Abstraction ([LZ75]) και HIPO ([STA76]) αποσύνθεταν τα προγράμματα κατά την φάση του Σχεδιασμού. Σε αντίθεση με τις προαναφερθέντες τεχνικές ο Τεμαχισμός Προγράμματος ([WEI84, TIP95, HH01, LUC01]) εφαρμόζεται σε προγράμματα που ήδη έχουν γραφεί. Αυτό τον καθιστά περισσότερο χρήσιμο για τεχνικές όπως η Συντήρηση από ότι ο Σχεδιασμός.

Ο Τεμαχισμός Προγράμματος είναι μια τεχνική για απλοποίηση προγραμμάτων επικεντρώνοντας το ενδιαφέρον σε επιλεγμένα κομμάτια του κώδικα. Κατά την διαδικασία του Τεμαχισμού Προγράμματος διαγράφονται τα μέρη του προγράμματος τα οποία δεν μας ενδιαφέρουν, λαμβάνοντας υπόψη το γεγονός πως υπάρχουν περιπτώσεις που μόνο ένα μέρος της λειτουργικότητας ενός προγράμματος είναι ενδιαφέρον.

Στη συνέχεια θα κάνουμε μια σύντομη ιστορική αναδρομή στον Τεμαχισμό Προγράμματος, βλέποντας τα κύρια σημεία στην εξέλιξη αυτής της τεχνικής. Αρχικά ο Τεμαχισμός Προγράμματος προτάθηκε από τον Weiser το 1979 ανάγκη για να βοηθήσει τους φοιτητές του να καταλάβουν και να αποσφαλματώσουν τα προγράμματα τους, ήταν το ερέθισμα για να αρχίσει την έρευνα του. Ο Weiser παρατήρησε ότι οι προγραμματιστές νοητικά χρησιμοποιούν τεμάχια προγράμματος (slices) κατά την αποσφαλμάτωση [WEI82] και προσπάθησε να ορίσει αυτή τη διαδικασία και τα αποτελέσματα της [WEI84]. Παρουσίασε λοιπόν μια προσέγγιση για να υπολογίσει τα τεμάχια (slices) ενός προγράμματος, στηριζόμενος στην επαναληπτική ανάλυση ροής στοιχείων. Για τον σκοπό αυτό χρησιμοποίησε γράφο ροής ελέγχου (control flow diagram) ως ενδιάμεση αντιπροσώπευση του δικού του αλγορίθμου τεμαχισμού προγράμματος.

Σύμφωνα λοιπόν με τον αρχικό ορισμό [WEI84], κατά τον Weiser ένα τεμάχιο προγράμματος (program slice) αποτελείται από εκείνα τα μέρη του προγράμματος που πιθανός να επηρεάζουν τις τιμές που υπολογίζονται σε κάποιο σημείο ενδιαφέροντος, το οποίο αναφέρεται ως κριτήριο τεμαχισμού (slicing criterion). Το κριτήριο τεμαχισμού είναι η πλειάδα <x,V> όπου x είναι ένα σημείο του προγράμματος και V είναι ένα υποσύνολο των μεταβλητών του προγράμματος. Τα μέρη του προγράμματος τα οποία επηρεάζουν άμεσα ή έμμεσα τις τιμές που υπολογίζονται στο κριτήριο C ονομάζονται τεμάχια προγράμματος (program slice) με βάση το κριτήριο C. Η διαδικασία του υπολογισμού φετών προγράμματος ονομάζεται τεμαχισμός προγράμματος (program slicing).

Στην πραγματικότητα ο τεμαχισμός προγράμματος που καθόρισε ο Weiser είναι ένα είδος εκτελέσιμου, στατικού τεμαχισμού οπισθοδρόμησης (executable backward static

slicing). Με τον όρο «εκτελέσιμο» εννοούμε ότι το τεμάχιο δεν είναι απλά ένα σύνολο εντολών αλλά μπορεί να μεταγλωττιστεί και να εκτελεστεί, «προς τα πίσω» (backward) λόγω της κατεύθυνσης που διαπερνούνται οι ακμές του γράφου εξάρτησης (dependence graph) κατά τη δημιουργία του τεμαχίου και «στατικός» διότι χρησιμοποιείται μόνο στατική πληροφορία για τον υπολογισμό των φετών προγράμματος, μιας και δεν λαμβάνονται υπόψη οι τιμές εισόδου του προγράμματος.

Την ιδέα του Weiser ακολούθησαν οι Korel και Laski οι οποίοι εισήγαγαν την έννοια του δυναμικού τεμαχισμού (dynamic slicing) [KL90]. Αντίθετα από τα στατικά τεμάχια, ένα δυναμικό τεμάχιο ενός προγράμματος υπολογίζεται με βάση μια συγκεκριμένη εκτέλεση του προγράμματος. Έτσι, τα δυναμικά τεμάχια (dynamic slices) που παράγονται είναι μικρότερες από τις στατικές (static slices), λόγω του ότι δεν περιλαμβάνουν τις γραμμές του προγράμματος που δεν σχετίζονται με το κριτήριο σε μία συγκεκριμένη τιμή εισόδου του προγράμματος. Αυτά τα δύο χαρακτηριστικά των δυναμικών φετών, το μικρότερο μέγεθος τους και το γεγονός ότι είναι σχετικές με μία εκτέλεση του προγράμματος τις κάνουν πιο ακριβής και συνεπώς περισσότερες βοηθητικές κατά την αποσφαλμάτωση. Όπως και ο Weiser, έτσι και ο Korel και Laski στον αλγόριθμο τεμαχισμού χρησιμοποιούν γράφο ροής ελέγχου ως ενδιάμεση αντιπροσώπευση.

Οι Agrawal και Horgan χρησιμοποίησαν γράφο ροής εξάρτησης (dependence graph) του προγράμματος για να επεκτείνουν την έννοια του δυναμικού τεμαχισμού (dynamic slicing), προτείνοντας τον σχετικό τεμαχισμό (relevant slicing) [AHKL93], [XZWC05]. Ένα σχετικό τεμάχιο ως προς μια μεταβλητή περιλαμβάνει, τόσο τις δηλώσεις που έχουν επιρροή στην μεταβλητή, όσο και τις εκτελέσιμες δηλώσεις που δεν είχαν επιπτώσεις στο αποτέλεσμα, αλλά θα μπορούσαν να έχουν επιπτώσεις αν αξιολογούνταν διαφορετικά. Αυτή η προσέγγιση διευκολύνει την αυξητική δοκιμή οπισθοδρόμησης (incremental regression testing).

Στην συνέχεια ο Venkatesh εισήγαγε το quasi στατικό τεμαχισμό (quasi-static slicing) [VE91]. Ο quasi στατικός τεμαχισμός είναι μια μέθοδος τεμαχισμού που συνδυάζει στοιχεία στατικού και δυναμικού τεμαχισμού και χρησιμοποιείται για την ανάλυση της

συμπεριφοράς ενός προγράμματος όταν καθορίζονται μερικές από τις μεταβλητές εισαγωγής.
Έπειτα ο Field πρότεινε το τεμαχισμό υπό περιορισμούς (constrained slicing) [FRT95], ο οποίος είναι παρόμοιος με τον quasi στατικό τεμαχισμό. Η μεγαλύτερη τους διαφορά είναι ότι η κατάσταση εισαγωγής ενός περιορισμένου τεμαχίου χαρακτηρίζεται από ένα κατηγόρημα στη γλώσσα προγραμματισμού και όχι ένα αρχικό πρόθεμα της ακολουθίας εισαγωγής.

Αργότερα ο Canfora πρότεινε τον τεμαχισμό υπό συνθήκη (conditioned slicing) [CCL98], ο οποίος αποτελεί μια πιο γενικευμένη μορφή του quasi στατικού τεμαχισμού και του τεμαχισμού υπό περιορισμούς (constrained slicing). Αυτή η μορφή τεμαχισμού επιτρέπει μια καλύτερη αποσύνθεση του προγράμματος δίνοντας έτσι τη δυνατότητα στους προγραμματιστές να αναλύσουν κομμάτια του κώδικα υπό διαφορετική οπτική σκοπιά.

Ο Hall εισήγαγε τον ταυτόχρονο δυναμικό τεμαχισμό (simultaneous dynamic slicing) [HA95], για τον υπολογισμό των φετών σε σχέση με ένα σύνολο εκτελέσεων προγράμματος. Το τελικό τεμάχιο που δημιουργείται με βάση τα δυναμικά τεμάχια που αντιστοιχούν σε κάθε εκτέλεση του συνόλου των εκτελέσεων του προγράμματος. Αυτή με μέθοδος τεμαχισμού χρησιμοποιείται για τον εντοπισμό της λειτουργικότητας του κώδικα.

Ο υβριδικός τεμαχισμός (hybrid slicing), διαφοροποιήθηκε από τις τεχνικές τεμαχισμού που είχαν προταθεί μέχρι στιγμής οι οποίες τεμαχίζουν ένα πρόγραμμα μόνο για ένα σύνολο εκτελέσεων. Ο υβριδικός τεμαχισμός λοιπόν ενσωματώνει και τις στατικές και τις δυναμικές πληροφορίες, χρησιμοποιώντας τη στατική πληροφορία για τη διεξαγωγή δυναμικού τεμαχισμού ή χρησιμοποιώντας τη δυναμική πληροφορία για τη διεξαγωγή στατικού τεμαχισμού.

Οι παραδοσιακές μέθοδοι τεμαχισμού βασίζονται όλες στη διαγραφή εντολών, αυτό τον περιορισμό απαλείφει ο άμορφος τεμαχισμός (amorphous slicing) ο οποίος προτάθηκε από τον Harman. Ο άμορφος τεμαχισμός έχοντας στη διάθεση του μια ευρεία γκάμα μετασχηματισμών, περιλαμβανομένου και τις διαγραφής εντολών, δίνει συχνά τεμάχια

προγράμματος με αρκετά πιο μικρό μέγεθος από αυτό που έχουν οι ισοδύναμα τεμάχιο στις οποίες διατηρείται η σύνταξη. Αυτή η τεχνική είναι ιδιαίτερα χρήσιμη στην κατανόηση προγράμματος.

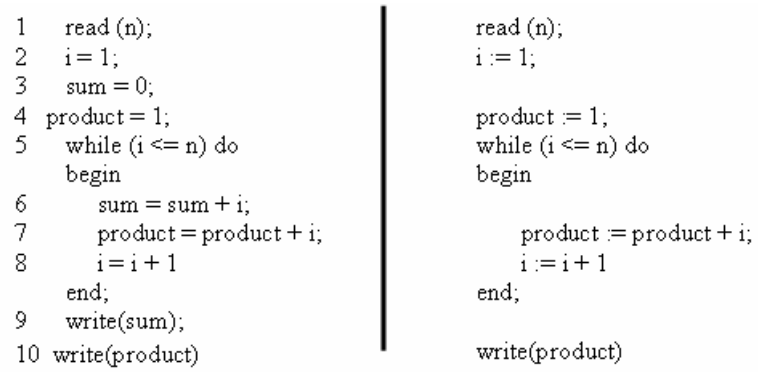
## 3.2 Τεχνικές Τεμαχισμού Προγράμματος

Στη διαδικασία τεμαχισμού προγράμματος διακρίνονται δύο βασικές διαστάσεις: η σημασιολογική (semantic) και η συντακτική (syntactic) [HH01]. Η σημασιολογική διάσταση ορίζει τι θα παραμείνει από το αρχικό πρόγραμμα ενώ η συντακτική διάσταση ορίζει αν θα διατηρηθεί η αρχική σύνταξη του προγράμματος αφαιρώντας απλά τις γραμμές εκείνες του κώδικα που δεν επηρεάζουν το σημείο υπολογισμού του προγράμματος που μας ενδιαφέρει ή αν επιτρέπεται οποιαδήποτε μετατροπή στον αρχικό κώδικα του προγράμματος. Ως προς την σημασιολογική διάσταση έχουν προταθεί πέντε βασικά παραδείγματα-τεχνικές: ο στατικός τεμαχισμός (static slicing), ο δυναμικός τεμαχισμός (dynamic slicing), ο quasi στατικός τεμαχισμός (quasi static slicing), ο ταυτόχρονος δυναμικός τεμαχισμός (simultaneous dynamic slicing) και ο τεμαχισμός υπό συνθήκη (conditioned slicing). Ως προς την συντακτική διάσταση διακρίνεται η τεχνική διατήρησης της σύνταξης (syntax-observing slice) και η τεχνική άμορφου τεμαχισμού (amorphous slicing).

Γενικά υπάρχουν δύο τεμάχια προγράμματος που μπορεί να δημιουργήσει κάποιος έχοντας επιλέξει το κριτήριο του προγράμματος, το προς τα πίσω τεμάχιο (backward slice) και το προς τα εμπρός τεμάχιο (forward slice). Το προς τα πίσω τεμάχιο αποτελείται από τις γραμμές εκείνες του κώδικα οι οποίες επηρεάζουν το κριτήριο που έχουμε επιλέξει, ενώ το προς τα εμπρός τεμάχια αποτελείται από τις γραμμές του κώδικα που επηρεάζονται από το κριτήριο που έχουμε επιλέξει. Σε αυτή τη μελέτη όταν αναφερόμαστε στον τεμαχισμό προγράμματος θα εννοούμε τεμαχισμό προς τα πίσω, όλες όμως οι τεχνικές τεμαχισμού που θα αναφερθούν έχουν τις ισοδύναμες τους τεχνικές τεμαχισμού προς τα εμπρός. Στη συνέχεια θα κάνουμε μια διεξοδική αναφορά στις κυριότερες τεχνικές τεμαχισμού.

### 3.2.1 Στατικός Τεμαχισμός (Static Slicing)

Στον στατικό τεμαχισμό ένα τεμάχιο προγράμματος (program slice) δημιουργείται με αφαίρεση των γραμμών εκείνων του κώδικα που δεν επηρεάζουν τον υπολογισμό της τιμής της μεταβλητής ή των μεταβλητών του ενδιαφέροντος στο συγκεκριμένο σημείο του κώδικα. Το τεμάχιο δηλαδή ενός προγράμματος P δημιουργείται με βάση ένα κριτήριο C=(x,V) όπου το x είναι ο αριθμός μιας συγκεκριμένης γραμμής κώδικα (statement) και V ένα σύνολο μεταβλητών που ορίζονται εντός του προγράμματος P. Ένα στατικό τεμάχιο προγράμματος περιέχει όλες τις γραμμές κώδικα που άμεσα ή έμμεσα επηρεάζουν την τιμή των μεταβλητών V στο συγκεκριμένο σημείο κώδικα για όλες τις πιθανές εισόδους του προγράμματος. Για τον υπολογισμό των φετών (slices) χρησιμοποιούνται μόνο στατικές πληροφορίες, για αυτό και η ονομασία στατικός τεμαχισμός.




**Εικόνα 3.1: (α) Αρχικός Κώδικας (β) Εφαρμογή στατικού τεμαχισμού με κριτήριο (10,product) [TIP95]**

Για να δούμε πως εφαρμόζεται ο στατικός τεμαχισμός ας πάρουμε το παράδειγμα [TIP95] της Εικόνα 3.1(α), υποθέτουμε ότι μας ενδιαφέρει μόνο η επίδραση που έχει ο πιο πάνω κώδικας στη μεταβλητή product στο τέλος του προγράμματος (γραμμή 10). Το αρχικό πρόγραμμα ζητά ένα αριθμό n και στη συνέχεια υπολογίζει το sum και το product των πρώτων n θετικών αριθμών. Όπως βλέπουμε και στην Εικόνα 3.1(β), έχουν αφαιρεθεί όλες οι γραμμές οι οποίες δεν επηρεάζουν το τελικό αποτέλεσμα της μεταβλητής product. Καθώς το τεμάχιο είναι πιο μικρό από το αρχικό πρόγραμμα, περιλαμβάνει όμως όλες τις εντολές οι οποίες μπορεί να επηρεάσουν την τιμή της μεταβλητής product, ο εντοπισμός του σφάλματος στο τεμάχχιο προγράμματος (slice)

είναι πιο γρήγορος από ότι στο αρχικό πρόγραμμα. Ωστόσο τα στατικά τεμάχια προγράμματος τείνουν να έχουν μεγάλο μέγεθος, αυτό ενισχύεται για καλά-δομημένα προγράμματα, στα οποία η συνεκτικότητα είναι μεγάλη, πράγμα που συνεπάγεται ότι ο υπολογισμός της τιμής μιας μεταβλητής εξαρτάται σε μεγάλο βαθμό από την τιμή πολλών άλλων μεταβλητών.

Στη βιβλιογραφία έχουν προταθεί διάφορες εφαρμογές του στατικού τεμαχισμού όπως η συντήρηση λογισμικού [GL91], ο έλεγχος [BI98, GHS95, HS91, HD95, HHD99], η αποσφαλμάτωση [WEI82, LW87], επικύρωση (validation) [KS98], συνένωση προγράμματος [HPR89], το reverse engineering και η κατανόηση προγράμματος [BE93, DKN01], η αναδόμηση προγράμματος (program restructuring) [KK94, GCLL00, CLM95, LD98, KB98] και ο εντοπισμός των επαναχρησιμοποιήσιμων συναρτήσεων [CLLF94, CLM96, LV97].

### 3.2.2 Δυναμικός τεμαχισμός (Dynamic Slicing)

Ένα δυναμικό τεμάχιο (dynamic slice) είναι το μέρος εκείνο του προγράμματος που επηρεάζει τον υπολογισμό της τιμής μίας μεταβλητής ή ενός συνόλου μεταβλητών κατά τη διάρκεια δυναμικού ελέγχου (dynamic testing) του λογισμικού για συγκεκριμένη είσοδο.

Τα κριτήρια που θα χρησιμοποιηθούν για τον υπολογισμό ενός δυναμικού τεμαχίου λαμβάνονται εάν προσθέσουμε στα κριτήρια που χρησιμοποιούνται για τον υπολογισμό ενός στατικού τεμαχίου, την είσοδο του προγράμματος. Να υπενθυμίσουμε ότι ένα στατικό τεμάχιο για ένα πρόγραμμα P δομείται με τη βοήθεια της πλειάδας (P,x,V), όπου x είναι ένα σημείο του προγράμματος, και V είναι ένα σύνολο μεταβλητών που ανήκουν στο P. Μία δήλωση μπορεί να αφαιρεθεί από το πρόγραμμα P και να μην αποτελέσει μέρος του static slice, εάν και μόνο εάν δεν μπορεί να επηρεάσει την τιμή της μεταβλητής στο V. Παρόμοια, ένα δυναμικό τεμάχιο υπολογίζεται, ξεχωρίζοντας εκείνες τις δηλώσεις του προγράμματος, που εκτελούνται μόνο για την είσοδο αυτή. Από αυτήν μάλιστα την άποψη, φαίνεται καλύτερα η βασική διαφορά από τη μέθοδο του στατικού τεμαχισμού, στην οποία συμπεριλαμβάνονται όλες οι πιθανές εκτελέσεις

για συγκεκριμένο κριτήριο και όχι μόνο αυτές που σχετίζονται με την είσοδο του προγράμματος [BDGHKK05].

Ένα δυναμικό τεμάχιο για ένα πρόγραμμα P δομείται με τη βοήθεια της πλειάδας (P, I, x, V) όπου P, x και V, συμβολίζουν αυτά που αναφέραμε πιο πάνω, ενώ I είναι η είσοδος του προγράμματος για την οποία θα προσδιοριστεί το δυναμικό τεμάχιο. Όλες οι δηλώσεις του προγράμματος που δεν προσπελαύνονται κατά τη διάρκεια της εκτέλεσης για τη συγκεκριμένη είσοδο I και εκείνες που δεν επηρεάζουν την τιμή των μεταβλητών στο V, όταν η επόμενη δήλωση που θα εκτελεστεί είναι στο σημείο x, αποκλείονται από το τεμάχιο s, του P.

```
(1)   read(n);                          read(n);
(2)   i := 1;                           i := 1;
(3)   while (i <= n) do                 while (i <= n) do
      begin                             begin
(4)     if (i mod 2 = 0) then             if (i mod 2 = 0) then
(5)       x := 17                           x := 17
        else                              else
(6)       x := 18;                                ;
(7)     i := i + 1                        i := i + 1
      end;                              end;
(8)   write(x)                          write(x)
```

**Εικόνα 3.2: (α) Αρχικός Κώδικας (β) Εφαρμογή δυναμικού τεμαχισμού με κριτήριο (n = 2,$8^1$,x) [TIP95]**

Στο παράδειγμα της Εικόνα 3.2(β) [TIP95] βλέπουμε πως εφαρμόζεται σε ένα πρόγραμμα (Εικόνα 3.2(α)) το δυναμικό κριτήριο (n=2, $8^1$, x). Το $8^1$ δηλώνει την πρώτη εμφάνιση του statement 8 στο όλο ιστορικό της εκτέλεσης του προγράμματος. Η είσοδος του προγράμματος, n=2, καθορίζει ότι το loop θα εκτελεστεί 2 φορές και οι αναθέσεις x:=17 και x:=18 θα εκτελεστούν από μία φορά. Στο παράδειγμα, την πρώτη φορά που θα εκτελεστεί το loop θα γίνει η ανάθεση του 18 στο x. Την δεύτερη όμως φορά το x θα γίνει 17. Οπόταν το x:=18 δεν επηρεάζει την τελική τιμή του x. Γι΄ αυτό και στο δυναμικό τεμάχιο αυτού του προγράμματος δεν συμπεριλαμβάνουμε τη δήλωση x:=18 (Εικόνα 3.2(β). Στο ίδιο παράδειγμα, το στατικό τεμάχιο του προγράμματος θα περιλάμβανε όλο το αρχικό πρόγραμμα.

Είναι λοιπόν φανερό ότι το μέγεθος ενός δυναμικού τεμαχίου σε σύγκριση με το μέγεθος του στατικού τεμαχίου ενός προγράμματος μπορεί να είναι αρκετά πιο μικρό, πράγμα που διευκολύνει την εύρεση σφαλμάτων.

Ο δυναμικός τεμαχισμός όπως προτάθηκε από τους Korel και Lasky [KL90] δημιουργεί εκτελέσιμα τεμάχια προγράμματος. Ωστόσο κατοπινοί αλγόριθμοι επιστρέφουν μη-εκτελέσιμα τεμάχια προγράμματος [AH90, GO91]. Η μη-εκτελεσιμότητα των φετών προγράμματος συμβάλει ακόμη περισσότερο στη μείωση του μεγέθους τους, πράγμα που τις καθιστά ακόμη πιο βοηθητικές για την αποσφαλμάτωση.

Εκτός από την αποσφαλμάτωση [KL90, KA98, ADS93], ο δυναμικός τεμαχισμός έχει χρησιμοποιηθεί σε πληθώρα εφαρμογών συμπεριλαμβανομένου του ελέγχου (software testing) [KFS93], της συντήρησης (software maintenance) [KO93, KR98] και της κατανόησης του προγράμματος (software comprehension) [KR97].

### 3.2.3 Quasi Στατικός Τεμαχισμός (Quasi Static Slicing)

Ο quasi στατικός τεμαχισμός (quasi-static slicing) [VE91] ήταν η πρώτη προσπάθεια να καθοριστεί μια υβριδική μέθοδος τεμαχισμού που κυμαίνεται μεταξύ του στατικού και δυναμικού τεμαχισμού. Η ανάγκη για την ανάλυση της συμπεριφοράς εφαρμογών στις οποίες μερικές από τις μεταβλητές εισόδου έχουν καθορισμένη τιμή, ενώ οι τιμές εισόδου αλλάζουν για τις υπόλοιπες μεταβλητές του προγράμματος, οδήγησε στη δημιουργία του quasi στατικού τεμαχισμού. Πράγματι, ένα quasi στατικό τεμάχιο διατηρεί τη συμπεριφορά του αρχικού προγράμματος όσον αφορά τις μεταβλητές του κριτηρίου τεμαχισμού, σε ένα υποσύνολο των πιθανών εισόδων προγράμματος. Αυτό το υποσύνολο διευκρινίζεται από τον πιθανό συνδυασμό τιμών. Φυσικά, στην περίπτωση όπου όλες οι μεταβλητές είναι χωρίς περιορισμούς, το quasi στατικό τεμάχιο συμπίπτει ένα στατικό τεμάχιο, ενώ στην αντίθετη περίπτωση δηλαδή όταν καθορίζονται οι τιμές όλων των μεταβλητών εισαγωγής, το quasi στατικό τεμάχιο είναι ένα δυναμικό τεμάχιο.

Ο quasi στατικός τεμαχισμός (quasi-static slicing) έχει εφαρμοστεί για την κατανόηση προγράμματος σε συνδυασμό με τους μετασχηματισμούς (transformations) [HDS95].

### 3.2.4 Ταυτόχρονος Δυναμικός Τεμαχισμός (Simultaneous Dynamic Slicing)

Μια διαφορετική μορφή τεμαχισμού που εισήγαγε ο Hall [HA95], υπολογίζει τα τεμάχια όσον αφορά ένα σύνολο εκτελέσεων του προγράμματος. Αυτή η μέθοδος τεμαχισμού ονομάζεται ταυτόχρονος δυναμικός τεμαχισμός προγράμματος, επειδή επεκτείνει το δυναμικό τεμαχισμό και τον εφαρμόζει ταυτόχρονα σε ένα σύνολο περιπτώσεων δοκιμής, παρά σε ένα μόνο σενάριο. Ένα ταυτόχρονο τεμάχιο προγράμματος σε ένα σύνολο περιπτώσεων δοκιμής δεν δίνεται απλά από την ένωση των δυναμικών φετών στις περιπτώσεις ελέγχου συστατικών (component test cases) αφού η ένωση δεν διατηρεί ακρίβεια σε όλες τις τιμές εισόδου. Ο Hall [HA95] πρότεινε έναν επαναληπτικό αλγόριθμο ο οποίος αρχίζοντας από ένα αρχικό σύνολο δηλώσεων συνεχίζει χτίζοντας επαυξητικά το ταυτόχρονο δυναμικό τεμάχιο, με τον υπολογισμό σε κάθε επανάληψη ενός μεγαλύτερου δυναμικού τεμαχίου.

Αυτή η τεχνική δημιουργίας φετών έχει χρησιμοποιηθεί για τον εντοπισμό της λειτουργικότητας μέσα στον κώδικα.

### 3.2.5 Τεμαχισμός υπό Συνθήκη (Conditioned Slicing)

Ο τεμαχισμός υπό συνθήκη είναι ένα γενικό πλαίσιο εργασίας για τεμαχισμό με βάση τη διαγραφή δηλώσεων [CCL98]. Ένα υπό συνθήκη τεμάχιο αποτελείται από ένα υποσύνολο των δηλώσεων του προγράμματος το οποίο διατηρεί τη συμπεριφορά του αρχικού προγράμματος με βάση ένα κριτήριο τεμαχισμού για κάθε σύνολο εκτελέσεων του προγράμματος. Ο τεμαχισμός υπό συνθήκη επιτρέπει μια καλύτερη αποσύνθεση του προγράμματος που δίνει στους ανθρώπινους αναγνώστες τη δυνατότητα να αναλύσουν τον κώδικα όσον αφορά διαφορετικές σκοπιές. Ένα τεμάχιο υπό συνθήκη μπορεί να υπολογιστεί σε δύο στάδια. Αρχικά γίνεται απλοποίηση του προγράμματος με βάση την συνθήκη εισόδου (για παράδειγμα την απόρριψη απραγματοποίητων

πορειών με βάση την συνθήκη είσοδο) και έπειτα γίνεται ο υπολογισμός ενός τεμαχίου με βάση στις δηλώσεις που έχουν απομείνει.

Ο τεμαχισμός υπό συνθήκη έχει εφαρμοστεί για την κατανόηση του προγράμματος [DFHH01, LFM96] καθώς και την εξαγωγή των επαναχρησιμοποιήσιμων συναρτήσεων [CCLL94, NEK93].

### 3.2.6 Άμορφος Τεμαχισμός (Amorphous Slicing)

Όλες οι τεχνικές τεμαχισμού που έχουν αναφερθεί μέχρι στιγμής διατηρούν την σύνταξη στο τεμάχιο προγράμματος. Αυτό ισχύει γιατί δημιουργούνται με μόνο μετασχηματισμό τη διαγραφή δηλώσεων, έτσι οι δηλώσεις που μένουν στο τεμάχιο είναι ένα συντακτικό υποσύνολο του αρχικού προγράμματος. Αντίθετα ο άμορφος τεμαχισμός ο οποίος προτάθηκε από τους Harman και Danicic [HD97], επιτρέπει κάθε μετασχηματισμό προγράμματος ο οποίος απλοποιεί το πρόγραμμα και διατηρεί το αποτέλεσμα του προγράμματος με βάση το κριτήριο τεμαχισμού . Όπως και στην περίπτωση ενός παραδοσιακού τεμαχίου προγράμματος, ένα άμορφο τεμάχιο διατηρεί τη σημασιολογία του αρχικού προγράμματος από το οποίο έχει προκύψει. Ωστόσο μπορεί να υπολογισθεί με εφαρμογή ενός μεγάλου φάσματος κανόνων μετασχηματισμού, συμπεριλαμβανομένου και της διαγραφής δηλώσεων.

Αυτή η τεχνική τεμαχισμού είναι ιδιαίτερα χρήσιμη για την κατανόηση του προγράμματος, καθώς με τους μετασχηματισμούς που εφαρμόζονται μπορούν να απλοποιηθούν σε μεγάλο βαθμό περίπλοκα προγράμματα [BHRS00]. Η εξαγωγή των επαναχρησιμοποιήσιμων συναρτήσεων είναι άλλη μια εφαρμογή του άμορφου τεμαχισμού.

## Κεφάλαιο 4

# Αλγόριθμοι Δυναμικού Τεμαχισμού Προγράμματος (Dynamic Program Slicing Algorithms)

---



---

Όπως έχει αναφερθεί και στο προηγούμενο κεφάλαιο, τα στατικά τεμάχια προγράμματος τείνουν να έχουν μεγάλο μέγεθος, γεγονός που ενισχύεται για καλά-δομημένα προγράμματα μιας και η συνεκτικότητα είναι μεγάλη. Ένα δυναμικό τεμάχιο προγράμματος είναι το μέρος του προγράμματος που επηρεάζει τη μεταβλητή που μας ενδιαφέρει για συγκεκριμένη εκτέλεση του προγράμματος, για αυτό έχει αρκετά μικρότερο μέγεθος από το αντίστοιχο στατικό τεμάχιο. Για το λόγο αυτό προτιμάται ο δυναμικός προγραμματισμός σε σχέση με τον στατικό προγραμματισμό, ειδικά για εφαρμογές όπως η αποσφαλμάτωση, η συντήρηση και ο έλεγχος προγράμματος.

## 4.1 Αλγόριθμος των Korel και Laski [KL90, TIP95]

Οι εισηγητές του δυναμικού προγραμματισμού, Korel και Laski [KL90], όρισαν το ιστορικό της εκτέλεσης του προγράμματος ως ένα μονοπάτι (trajectory) το οποίο αποτελείται από μία ακολουθία στιγμιότυπων δηλώσεων (statements) και κατηγορημάτων ελέγχου (control predicates). Για τον διαχωρισμό μεταξύ διαφορετικών στιγμιότυπων στο ιστορικό της εκτέλεσης γίνεται χρήση των ετικετών (labels). Στον αλγόριθμο των Korel και Laski το δυναμικό κριτήριο τεμαχισμού ορίζεται ως μια τριάδα $(x, I^q, V)$, όπου x δηλώνει την είσοδο του προγράμματος, $I^q$ το στιγμιότυπο της δήλωσης που βρίσκεται στο μονοπάτι του ιστορικού της εκτέλεσης και V ένα υποσύνολο των μεταβλητών του προγράμματος. Το δυναμικό τεμάχιο ορίζεται με βάση ένα κριτήριο $(x, I^q, V)$ ως ένα εκτελέσιμο πρόγραμμα S το οποίο παράγεται από ένα πρόγραμμα P με την αφαίρεση μηδέν ή περισσοτέρων δηλώσεων.

Το εκτελέσιμο πρόγραμμα που παράγεται S πρέπει να ικανοποιεί τους πιο κάτω περιορισμούς:

1. Όταν εκτελείται με είσοδο x το μονοπάτι εκτέλεσης του S είναι το ίδιο με το μονοπάτι εκτέλεσης του αρχικού προγράμματος P εάν αφαιρέσουμε τις δηλώσεις που δεν υπάρχουν στο τεμάχιο.
2. Το πρόγραμμα και το δυναμικό τεμάχιο επιστρέφουν τις ίδιες τιμές για όλες τις μεταβλητές του V για το στιγμιότυπο της δήλωσης που περιέχεται στο κριτήριο.

3. Τέλος είναι απαραίτητο κάθε δήλωση `I` της οποίας το στιγμιότυπο της ορίζεται από το κριτήριο τεμαχισμού, να υπάρχει στο τεμάχιο.

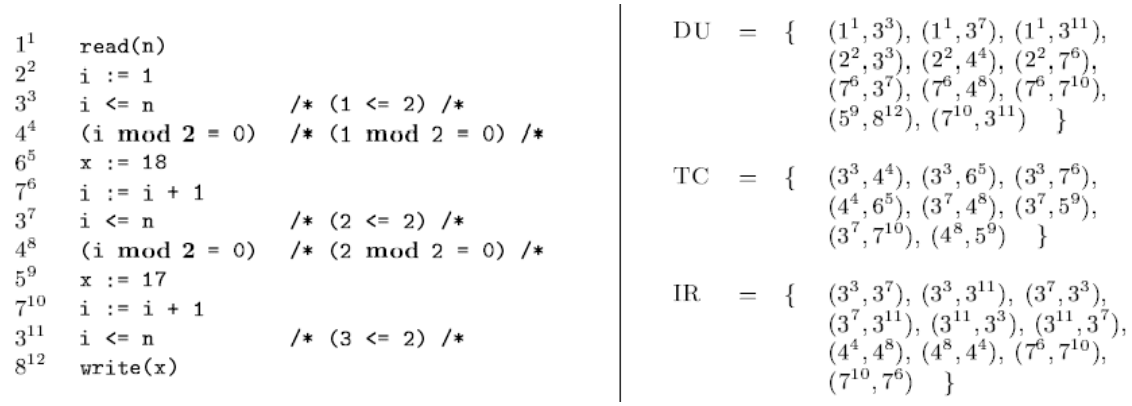
Οι συσχετίσεις μεταξύ στιγμιότυπων δηλώσεων που βρίσκονται μέσα στο μονοπάτι εκτέλεσης ορίζονται βάσει των πιο κάτω εννοιών Δυναμικής ροής (Dynamic flow):

• Defin ition Use (DU) είναι η σχέση μεταξύ μία μεταβλητής με τον προηγούμενο ορισμό της, ο οποίος είναι μοναδικός στο ιστορικό της εκτέλεσης.

• Test Control (TC) είναι η σχέση της πιο πρόσφατης ύπαρξης ενός κατηγορήματος ελέγχου (control predicate) με τα στιγμιότυπα της δήλωσης που ελέγχονται από αυτό στο ιστορικό της εκτέλεσης.

• Identity Relation (IR) είναι η σχέση που συνδέει τα στιγμιότυπα της ίδιας δήλωσης.

Ένα παράδειγμα για τις πιο πάνω σχέσεις είναι σε περίπτωση που έχουμε το μονοπάτι του ιστορικού που ακολουθεί



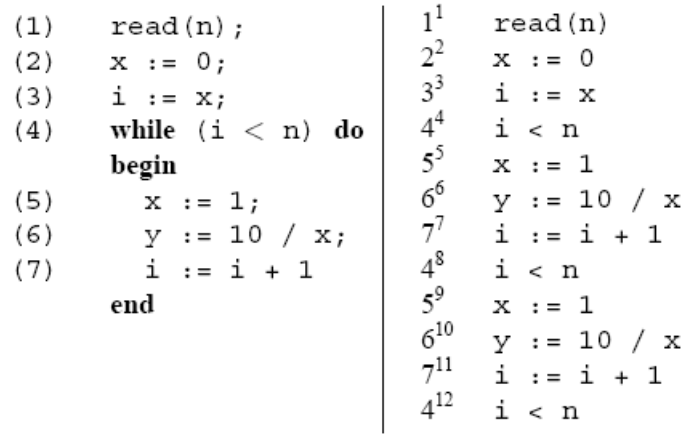

**Εικόνα 4.3: (α) Μονοπάτι (trajectory) για το παράδειγμα της Εικόνας 4.2 (β) Έννοιες Δυναμικής Ροής για το μονοπάτι εκτέλεσης [TIP95]**

Τα τεμάχια προγράμματος (slices) υπολογίζονται με επαναληπτικό τρόπο. Ο υπολογισμός αυτός γίνεται με την δημιουργία συνόλων δηλώσεων οι οποίες σχετίζονται είτε έμμεσα είτε άμεσα. Για ένα συγκεκριμένο κριτήριο τεμαχισμού `(x, I`$^{q}$`, V)` το αρχικό σύνολο αποτελείται από τους τελευταίους ορισμούς των μεταβλητών του `V` στο μονοπάτι εκτέλεσης πριν από το στιγμιότυπο της δήλωσης `I`$^{q}$. Περιέχει επίσης τις

εκτελέσεις που βρίσκονται στην διαδρομή της οποίας το στιγμιότυπο είναι control depended.

Για παράδειγμα στην περίπτωση του μονοπατιού του ιστορικού εκτέλεσης της Εικόνας 4.1 (α) και για κριτήριο τεμαχισμού: $(n=2, 8^{12}, \{x\})$, το αρχικά τεμάχιο θα περιέχει μόνο τη δήλωση $5^9$ αφού εκείνη μόνο σχετίζεται έμμεσα με τη δήλωση $8^{12}$: $A^0 = \{5^9\}$ Οι επόμενες επαναλήψεις θα δημιουργήσουν τα παρακάτω τεμάχια προγράμματος:
$A^1 = \{3^7, 4^8\}$, $A^2 = \{7^6, 1^1, 3^3, 3^{11}, 4^4\}$ και $A^3 = \{2^2, 7^{10}\}$

Οι σχέσεις DU και TC διασχίζουν τη διαδρομή μόνο με κατεύθυνση προς τα πίσω (backward). Ο σκοπός της IR σχέσης είναι να διασχίσει την διαδρομή και στις δύο κατευθύνσεις και να συμπεριλάβει όλες τις δηλώσεις και τα κατηγορήματα ελέγχου (control predicates) τα οποία είναι απαραίτητα να επιβεβαιώνουν τον τερματισμό των βρόγχων (loops) στα τεμάχια προγράμματος (slice). Ωστόσο δεν υπάρχει απόδειξη ότι η χρήση της IR σχέσης είναι πάντα επαρκής για τη δημιουργία φετών προγράμματος που όταν εκτελεστούν τερματίζουν. Εικάζουμε ότι ο περιορισμός της IR σχέσης σε στιγμιότυπα δηλώσεων που αναφέρονται σε κατηγορήματα ελέγχου (control predicates) στο πρόγραμμα θα έχει ως αποτέλεσμα την δημιουργία μικρότερων φετών προγράμματος (slices). Ένα δευτερεύον πρόβλημα που υπάρχει με τον συγκεκριμένο αλγόριθμο είναι ότι στην περίπτωση που δεν συμπεριληφθούν στο τελικό τεμάχιπ (slice) όλες οι δηλώσεις από τις οποίες εξαρτάται το κριτήριο τεμαχισμού τότε, υπάρχει μεγάλη πιθανότητα να έχουμε διαίρεση με το μηδέν. Για παράδειγμα για το κομμάτι κώδικα της Εικόνα 4.2 (α), το μονοπάτι εκτέλεσης του προγράμματος για είσοδο n=2 φαίνεται στην Εικόνα 4.2 (β).




**Εικόνα 4.4: (α) Αρχικός κώδικας (β) Μονοπάτι (trajectory) [TIP95]**

Στο τεμάχιο προγράμματος της Εικόνα 4.3. Παρατηρούμε ότι καθώς η δήλωση `x := 1` (γραμμή 5 στον αρχικό κώδικα) δεν συμπεριλαμβάνεται στο τεμάχιο προγράμματος, όταν αυτή εκτελεστεί με είσοδο `n = 2`, θα οδηγήσει σε διαίρεση με το μηδέν όταν θα εκτελεστεί η πράξη `y=10/x` (γραμμή 6 στον αρχικό κώδικα).

```
read(n);
x := 0;
i := x;
while (i < n) do
begin

  y := 10 / x;
  i := i + 1
end
```

**Εικόνα 4.5: Δυναμικό τεμάχιο για κριτήριο τεμαχισμού (n=2,6[10],{i}) [TIP95]**

## 4.2 Αλγόριθμος με Δυναμικές Σχέσεις εξαρτήσεων (Dynamic dependence relations)

Ο αλγόριθμος του Gopal [GO91, TIP95] υπολογίζει τα δυναμικά τεμάχια προγράμματος με χρήση των δυναμικών σχέσεων εξαρτήσεων (dynamic dependence relations). Βασίζεται στον αλγόριθμο των Bergeretti και Carrıe information-flow relations αλλά στην δυναμική του έκδοση.

Οι σχέσεις αυτές ορίζονται ως εξής:

- Η λ σχέση περιέχει όλα τα ζεύγη (v , e) όπου η δήλωση (statement) e εξαρτάται από την τιμή εισόδου της μεταβλητής v κατά την εκτέλεση του συγκεκριμένου προγράμματος.
- Η μ σχέση περιέχει όλα τα ζεύγη (e , v) όπου η τελική τιμή του v εξαρτάται από την εκτέλεση της δήλωσης (statement) e.
- Η σχέση ρ περιέχει τα ζεύγη (v , v΄) όπου η τιμή εξόδου του v΄ εξαρτάται από την τιμή εισόδου του v.

Για κενές δηλώσεις (statements), αναθέσεις και ακολουθίες δηλώσεων οι εξαρτημένες σχέσεις (dependence relations) του Gopal είναι οι ίδιες με την περίπτωση του στατικού τεμαχισμού. Οι στατικές σχέσεις ροής-πληροφορίας (information-flow relations) για μια conditional δήλωση π.χ. if statement, εξάγονται από την ίδια τη δήλωση και τις δηλώσεις που αποτελούν τις διακλαδώσεις της δήλωσης. Ωστόσο όταν αναφερόμαστε σε σχέσεις δυναμικών εξαρτήσεων (dynamic dependence relations), μόνο οι εξαρτήσεις οι οποίες προέρχονται από την διακλάδωση η οποία εκτελείται λαμβάνονται υπόψη. Η σχέση εξάρτησης (dependence relation) για μια while δήλωση (statement), εκφράζεται υπό την μορφή εξαρτώμενων σχέσεων (dependence relations) για φωλιασμένα conditionals που έχουν την ίδια συμπεριφορά.

Για ορισμένες περιπτώσεις ο αλγόριθμος, του Gopal μπορεί δώσει ένα τεμάχια (slice) το οποίο δεν τερματίζει ποτέ για ένα πρόγραμμα το οποίο τερματίζει στην αρχική του μορφή. Ένα πλεονέκτημα της χρήσης των σχέσεων εξαρτήσεων (dependence relations) είναι ότι για ορισμένες περιπτώσεις, υπολογίζονται μικρότερα τεμάχια προγράμματος (slices) σε σύγκριση με τον αλγόριθμο των Korel και Laski.

## 4.3 Αλγόριθμος Miller και Choi

Οι Miller και Choi [CMN91, TIP95] ήταν οι πρώτοι που πρότειναν την δυναμική παραλλαγή του Γράφου Εξάρτησης του Προγράμματος (PDG), και την ονόμασαν

δυναμικός γράφος εξάρτησης του προγράμματος (dynamic program dependence graph). Αυτοί οι γράφοι χρησιμοποιούνται από το παράλληλο πρόγραμμα του debugger τους για εκτέλεση flowback ανάλυσης και κατασκευάζονται επαυξητικά κατά απαίτηση (on demand). Πριν από την εκτέλεση, κατασκευάζεται ένα στατικό PDG, και ο κώδικας αντικειμένου του προγράμματος αυξάνεται με τον κώδικα που παράγει ένα log αρχείο. Επιπρόσθετα παράγεται το emulation package. Τα προγράμματα χωρίζονται σε emulation blocks (στην ουσία υπορουτίνες). Κατά την διάρκεια της αποσφαλμάτωσης, ο debugger χρησιμοποιεί το log αρχείο, τον στατικό PDG και το emulation package με σκοπό την επανεκτέλεση του emulation block. Έτσι λαμβάνει τις πληροφορίες που είναι απαραίτητες για να κατασκευάσουν το μέρος του δυναμικού PDG που αντιστοιχεί σε εκείνο το block. Σε περίπτωση που ο χρήστης θέλει να εκτελέσει flowback analysis στα μέρη του γράφου που δεν έχουν κατασκευαστεί ακόμα θα πρέπει να επανεκτελέσει περισσότερα emulation blocks.

## 4.4 Αλγόριθμος Agrawal και Horgan

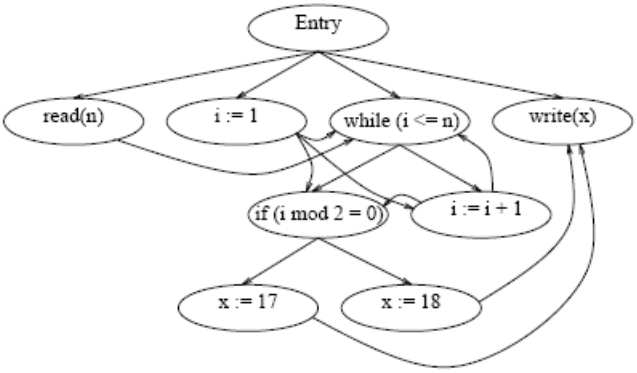
Οι Agrawal και Horgan παρουσίασαν τέσσερις προσεγγίσεις για τον υπολογισμό δυναμικών φετών [AH90, KR98b, TIP95] με χρήστη γράφων εξάρτησης (dependence graphs) - PDG. Ο γράφος εξάρτησης περιγράφει τις εξαρτήσεις δεδομένων και ελέγχου του προγράμματος. Κατά την εκτέλεση του προγράμματος οι ακμές του γράφου εξάρτησης που συναντώνται κατά την εκτέλεση του προγράμματος μαρκάρονται. Μετά την εκτέλεση του προγράμματος, ο αλγόριθμος διασχίζει τον γράφο εξάρτησης κατά μήκος των ακμών που έχουν μαρκαριστεί για να κτίσει το δυναμικό τεμάχιο.

Οι δύο πρώτοι αλγόριθμοι που έχουν προτείνει είναι απλοί και ανακριβείς. Μαρκάρουν κάποιο μέρος ενός PDG κατά την εκτέλεσή του. Ο πρώτος συγκεκριμένα μαρκάρει εκτελέσιμους κόμβους και ο δεύτερος εκτελέσιμες ακμές. Το μειονέκτημα και των δύο αλγορίθμων είναι ότι επικεντρώνονται σε ένα μέρος, ο ένας στους κόμβους και ο άλλος στις ακμές κάθε χρονική στιγμή που μια δήλωση (statement) ή μία εξαρτώμενη ακμή (dependence edge) μεταξύ δύο δηλώσεων εκτελείται.

### 4.4.1 Αλγόριθμος 1

Ο πρώτος αλγόριθμος αρχικά μαρκάρει όλους του κόμβους στον PDG σαν μη εκτελέσιμους. Κατά την εκτέλεση του προγράμματος κάθε φορά που εκτελείται μια δήλωση ο αντίστοιχος κόμβος της μαρκάρεται σαν εκτελέσιμος. Με το τέλος της εκτέλεσης το τεμάχιο (slice) υπολογίζεται με την εφαρμογή στατικού αλγορίθμου τεμαχισμού (static slicing algorithm) για τους κόμβους που είναι μαρκαρισμένοι σαν εκτελέσιμοι και τις ακμές που τους ενώνουν. Κατά συνέπεια, μη εκτελέσιμοι κόμβοι δεν θα συμπεριληφθούν στο δυναμικό τεμάχιο (dynamic slice).

Ο αλγόριθμος μαρκάρει μια ακμή χωρίς απαραίτητα να πρέπει να έχει επίδραση στον υπολογισμό της συνάρτησης ενδιαφέροντος. Ας πάρουμε πάλι το παράδειγμα της Εικόνας 4.2(α), ο PDG του εν λόγω παραδείγματος φαίνεται στην Εικόνα 4.4 πιο κάτω.




**Εικόνα 4.6: PDG του προγράμματος της Εικόνας 4.2(α) [TIP95]**

Ας υποθέσουμε ότι θέλουμε να υπολογίσουμε το τεμάχιο προγράμματος με κριτήριο τη τελική τιμή της μεταβλητής `x` για είσοδο `n = 2`. Μιας και όλες οι κορυφές του PDG εκτελούνται, μαρκάρονται όλες. Παρατηρούμε ότι ο αλγόριθμος θα συμπεριλάβει τη δήλωση `x := 18` στο τελικό τεμάχιο λόγω της ύπαρξης μιας ακμής από την κορυφή `x := 18` στην κορυφή `write(x)`, παρόλο που η ακμή αυτή δεν αντιπροσωπεύει εξάρτηση που συνέβη στη δεύτερη επανάληψη του κόμβου. Με τον πιο πάνω τρόπο μπορεί μερικές φορές να επιλεγούν δηλώσεις οι οποίες δεν επηρεάζουν την μεταβλητή που μας ενδιαφέρει έτσι αυτός ο αλγόριθμος δεν οδηγεί πάντα σε ακριβή τεμάχια προγράμματος.

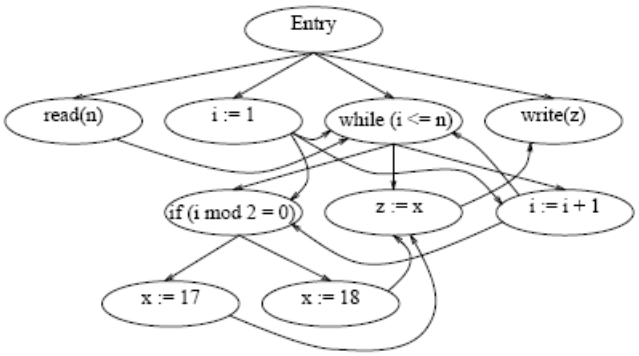
```
(1)    read(n);
(2)    i := 1;
(3)    while (i <= n) do
       begin
(4)      if (i mod 2 = 0) then
(5)        x := 17
         else
(6)        x := 18;
(7)      z := x;
(8)      i := i + 1
       end;
(9)    write(z)
```

**Εικόνα 4.7: Αρχικός κώδικας [TIP95]**

### 4.4.2 Αλγόριθμος 2

Ο δεύτερος αλγόριθμος μαρκάρει όλες τις εκτελέσιμες ακμές του PDG με βάση τις εξαρτήσεις που προκύπτουν κατά την εκτέλεση. Πάλι το τεμάχιο προγράμματος δημιουργείται διασχίζοντας τον PDG, με τη διαφορά ότι αυτή τη φορά μόνο οι μαρκαρισμένες ακμές συμπεριλαμβάνονται. Για το παράδειγμά μας η εξάρτηση υπάρχει μόνο στις επαναλήψεις του κόμβου για τις οποίες η μεταβλητή i έχει τιμή μονό αριθμό. Αυτό έχει σαν αποτέλεσμα να παίρνουμε ακριβέστερες απαντήσεις διότι αγνοούνται οι δηλώσεις οι οποίες εκτελέστηκαν, όμως δεν επηρέασαν τις τιμές των μεταβλητών στο τεμάχιο.

Στην Εικόνα 4.6 βλέπουμε το PDG του κώδικα της Εικόνα 4.5. Για είσοδο `n = 2`, όλες οι ακμές θα μαρκαριστούν και έτσι το τεμάχιο προγράμματος θα είναι όλο το πρόγραμμα.



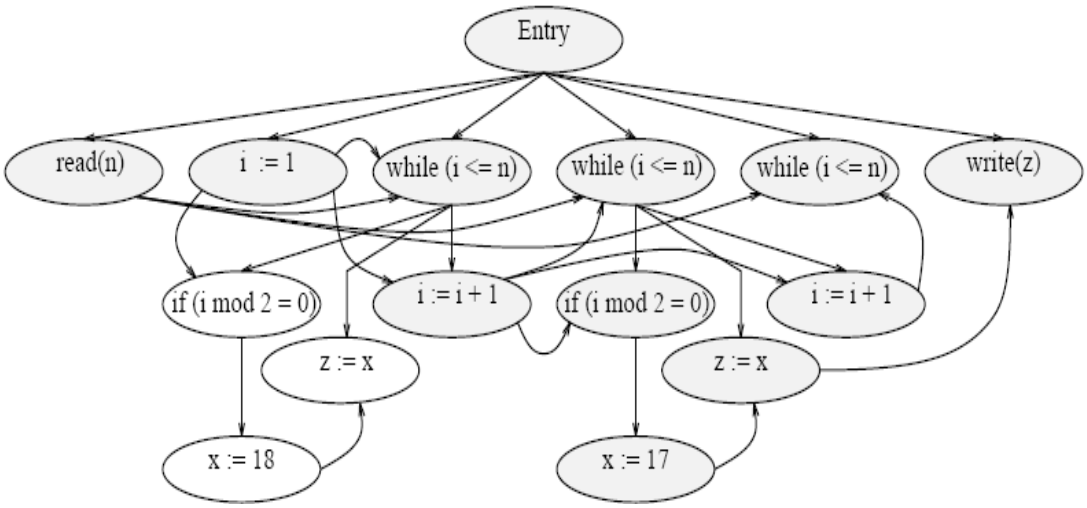

**Εικόνα 4.8: PDG του προγράμματος της Εικόνας 4.5 [TIP95]**

Όπως είδαμε και στο πιο πάνω παράδειγμα το μειονέκτημα αυτού του αλγορίθμου είναι ότι υπολογίζει τεμάχια προγράμματος τα οποία έχουν μεγάλο μέγεθος, μιας και δεν λαμβάνει υπόψη το γεγονός ότι διαφορετικά στιγμιότυπα μιας δήλωσης στο ιστορικό της εκτέλεσης μπορεί να είναι μεταβατικά εξαρτημένα από διαφορετικές δηλώσεις.

### 4.4.3 Αλγόριθμος 3 - Dynamic Dependence Graph

Ο τρίτος αλγόριθμος που προτάθηκε προσπαθεί να βελτιώσει τον δεύτερο με την δημιουργία ενός ξεχωριστού κόμβου στον γράφο εξάρτησης (dependence graph) για κάθε στιγμιότυπο της δήλωσης στο ιστορικό της εκτέλεσης. Κάθε κόμβος έχει εξαρτώμενες ακμές εξόδου σε αυτές μόνο τις δηλώσεις από τις οποίες το συγκεκριμένο στιγμιότυπο της δήλωσης εξαρτάται. Αυτού του είδους ο γράφος ονομάζεται Δυναμικός Γράφος Εξάρτησης (Dynamic Dependence Graph (DDG)). Ένα δυναμικό κριτήριο τεμαχισμού προσδιορίζεται από ένα κόμβο στο ν DDG, και ένα δυναμικό τεμάχιο υπολογίζεται με τον καθορισμό όλων των DDG κόμβων οι οποίοι ικανοποιούν το κριτήριο. Μια δήλωση ή ένα κατηγόρημα ελέγχου συμπεριλαμβάνεται στο τεμάχιο εάν το κριτήριο μπορεί να επιτευχθεί από τουλάχιστον ένα κόμβο από τα στιγμιότυπά του.

Στην Εικόνα 4.7 βλέπουμε το DDG για το παράδειγμα της Εικόνα 4.5. Το κριτήριο τεμαχισμού αντιστοιχεί στον κόμβο: `write(z)`, όλοι οι κόμβοι από τους οποίους μπορεί να επιτευχθεί το κριτήριο έχουν σκιαστεί. Παρατηρούμε ότι το κριτήριο δεν μπορεί να επιτευχθεί από τον κόμβο `x := 18`, για αυτό και δεν συμπεριλαμβάνεται στο τεμάχιο προγράμματος (slice).




**Εικόνα 4.9: DDG του προγράμματος της Εικόνας 4.5 [TIP95]**

Το μειονέκτημα της χρήσης DDG είναι ότι ο αριθμός των κόμβων σε ένα DDG είναι ίσος με τον αριθμό των εκτελεσμένων δηλώσεων, ο οποίος πιθανόν να εξαρτάται από την είσοδο εκτέλεσης, άρα ο αριθμός των κόμβων που χρειάζεται να αναλυθούν είναι απεριόριστος.

### 4.4.4 Αλγόριθμος 4 - Reduced Dynamic Dependence Graph

Ο τέταρτος αλγόριθμος προτάθηκε για την αντιμετώπιση του μειονεκτήματος του τρίτου αλγορίθμου. Για να μειωθεί ο αριθμός των κόμβων στο DDG γίνεται συγχώνευση κόμβων για τους οποίους οι μεταβατικές εξαρτήσεις αναφέρονται στο ίδιο σύνολο δηλώσεων, με άλλα λόγια ένας νέος κόμβος εισάγεται μόνο εάν μπορεί να δημιουργήσει ένα δυναμικό τεμάχιο προγράμματος. Σίγουρα αυτός ο έλεγχος στοιχίζει σε χρόνο. Ο γράφος που δημιουργείται ονομάζεται Reduced Dynamic Dependence Graph (RDDG). Τα τεμάχια που υπολογίζονται με την χρήση του RDDGs έχουν την ίδια ακρίβεια με εκείνες που υπολογίζονται με την χρήση DDGs.

Στην Εικόνα 4.7 οι κορυφές `i := i +1` και οι δύο κόμβοι `while(i <= n)` που φαίνονται δεξιά στην εικόνα, έχουν τις ίδιες μεταβατικές εξαρτήσεις, αυτοί οι κόμβοι εξαρτώνται από τις δηλώσεις `1`, `2`, `3` και `8` της Εικόνα 4.5. Έτσι ο RDDG για αυτό το πρόγραμμα, με είσοδο `n = 2`, παράγεται αν συνενώσουμε αυτούς τους τέσσερις DDG κόμβους σε ένα κόμβο.

Οι Agrawal και Horgan παρουσίασαν ένα αλγόριθμο για την κατασκευή του RDDG χωρίς να πρέπει να λάβει υπόψη ολόκληρο το ιστορικό της εκτέλεσης. Χρειάζεται μόνο για κάθε μεταβλητή, τον κόμβο που αντιστοιχεί στον τελευταίο ορισμό της, για κάθε κατηγόρημα τον κόμβο που αντιστοιχεί στην τελευταία εκτέλεσή του, και για κάθε κόμβο στον RDDG, το δυναμικό τεμάχιο του κόμβου.

Αυτή η προσέγγιση σε μερικές περιπτώσεις παράγει απλά ένα σύνολο δηλώσεων αντί ένα εκτελέσιμο τεμάχιο.

Στα επόμενα χρόνια έγιναν διάφορες προσπάθειες βελτίωσης των αποτελεσμάτων των πιο πάνω αλγορίθμων, θα δούμε επιγραμματικά μερικές από αυτές.

Οι Goswami και Mall [GM02, XZWC05] πρότειναν ένα αλγόριθμο δυναμικού τεμαχισμού βασισμένο στην έννοια του Compact Dynamic Dependence Graph (CDDP). Οι ακμές των εξαρτήσεων ελέγχου (control dependences) του CDDG δημιουργούνται στατικά ενώ οι εξαρτήσεις ροής-δεδομένων (data-flow dependences) δημιουργούνται δυναμικά. Αυτή η προσέγγιση είναι πιο αποτελεσματική σε χρόνο και χώρο από ότι οι RDDG-προσεγγίσεις.

Οι Mund, Mall και Sarkar [MMS03, XZWC05] εισήγαγαν την έννοια του Modified Program Dependence Graph (MPDG) σαν ενδιάμεση αναπαράσταση του προγράμματος, ο οποίος είναι ένα PDG με την προσθήκη σταθερών και ασταθών ακμών.

## 4.5 Ακριβείς αλγόριθμοι Δυναμικού Τεμαχισμού (Precise Dynamic Slicing Algorithms)

Οι Zhang, Gupta και Zhang [ZGZ03, XZWC05] παρουσίασαν τρεις ακριβείς αλγορίθμους τους οποίους θα δούμε με περισσότερη λεπτομέρεια στα επόμενα υποκεφάλαια. Οι αλγόριθμοι αυτοί είναι: ο αλγόριθμος Πλήρους Επεξεργασίας (Full Processing Algorithm-FP), ο αλγόριθμος καμίας επεξεργασίας (No Processing Algorithm-NP) και ο αλγόριθμος Περιορισμένης Επεξεργασίας (Limited Processing Algorithm-LP).

Πριν προχωρήσουμε στην περιγραφή των επιμέρους αλγορίθμων θα ήταν καλό να διευκρινίσουμε κάποιες σχετικές έννοιες. Η βασική προσέγγιση του δυναμικού τεμαχισμού προγράμματος περιλαμβάνει τα πιο κάτω βήματα. Αρχικά εκτελείται το πρόγραμμα μια φορά και παράγεται το ίχνος εκτέλεσης (execution trace) από το οποίο στη συνέχεια θα παραχθεί το DDG. Τέλος διασχίζοντας το DDG παίρνουμε το δυναμικό τεμάχιο προγράμματος. Το ίχνος εκτέλεσης (execution trace) περιέχει όλες τις πληροφορίες που σχετίζονται με το χρόνο εκτέλεσης του προγράμματος και οι οποίες μπορούν να χρησιμοποιηθούν από τους αλγόριθμους δυναμικού τεμαχισμού. Για αυτό το λόγο αναφέρεται και σαν ίχνος πλήρους ελέγχου ροής (full control flow trace) και επίσης σαν ίχνος αναφοράς μνήμης (memory reference trace). Επομένως, με τη βοήθεια του ίχνους εκτέλεσης, ξέρουμε την ακριβή πορεία του προγράμματος κατά τη διάρκεια της εκτέλεσης και επίσης μέσω δεικτών τις διευθύνσεις των δεδομένων που προσπελαύνονται.

Οι ακριβείς αλγόριθμοι Δυναμικού Τεμαχισμού ουσιαστικά αφορούν δύο διεργασίες: την προεπεξεργασία (preprocessing) που χτίζει έναν εξαρτώμενο γράφο με την ανάκτηση των δυναμικών εξαρτήσεων από το ίχνος εκτέλεσης του προγράμματος και τον τεμαχισμό (slicing) που υπολογίζει τα τεμάχια με βάση τα δεδομένα το ψ διασχίζοντας το δυναμικό γράφο εξάρτησης.

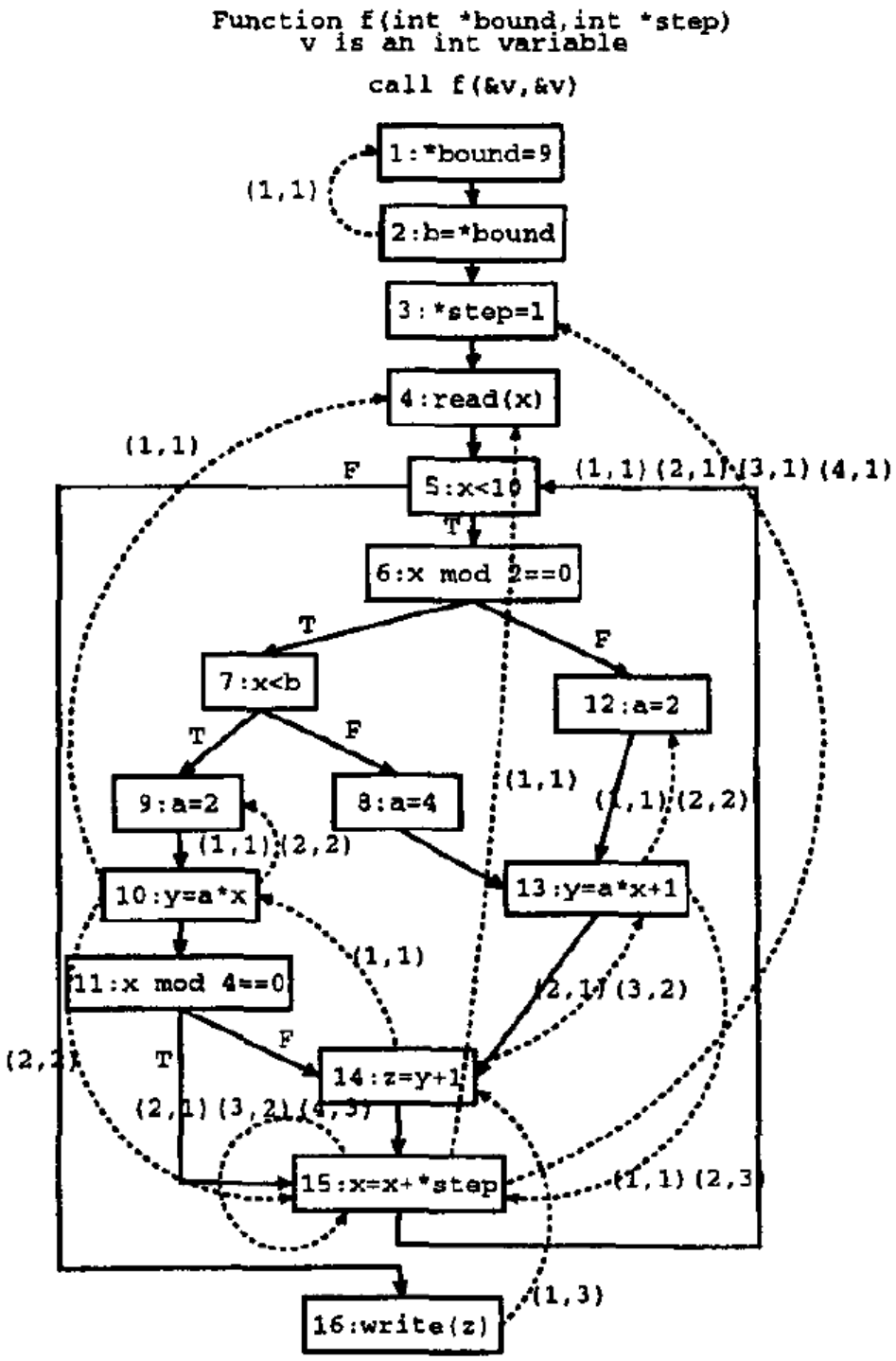
Οι αλγόριθμοι που παρουσιάζουμε, διαφέρουν κυρίως ως προς το βαθμό της προ-επεξεργασίας πριν τον υπολογισμό των δυναμικών φετών.

### 4.5.1 Αλγόριθμος Πλήρους Επεξεργασίας (Full Processing Algorithm-FP)

Περιεκτικά ο αλγόριθμος Πλήρους Επεξεργασίας (FP) κατά πρώτον προεπεξεργάζεται το ίχνος εκτέλεσης και προσθέτει μαρκαρισμένες ακμές στον DG. Κατά τον τεμαχισμό χρησιμοποιούνται οι ‘ταμπέλες’ αυτές για να διασχίσουμε μόνο τις σχετικές ακμές.

Στο παράδειγμα πιο κάτω (Εικόνα 4.8) φαίνονται οι δυναμικές ακμές εξάρτησης στοιχείων για ένα συγκεκριμένο τρέξιμο, με είσοδο `x = 6`. Οι ακμές μαρκάρονται με τα στιγμιότυπα των εκτελέσεων των μεταβλητών που περιλαμβάνονται στις εξαρτήσεις

στοιχείων. Το ακριβές δυναμικό τεμάχιο για την τιμή του z, που χρησιμοποιήθηκε για την εκτέλεση της δήλωσης 16 δίνεται από τη διάσχιση του γράφο υ Οι ακμές εξάρτησης στοιχείων κατά τη διάρκεια του υπολογισμού του τεμαχίου, περιλαμβάνουν τις : $(16_1,14_3)$, $(14_3,13_2)$, $(13_2,12_2)$, $(13_2,15_3)$, $(15_3,3_1)$, $(15_3,15_2)$, $(15_2,3_1)$, $(15_2,15_1)$, $(15_1,3_1)$, $(15_1,4_1)$




**Εικόνα 4.10: DDG του προγράμματος της Εικόνας 4.5 [ZGZ03]**

### 4.5.2 Αλγόριθμος Καμίας Επεξεργασίας (No Processing Algorithm-NP)

Ο αλγόριθμος FP πραγματοποιεί αρχικά όλη την προεπεξεργασία και μετά αρχίζει τον τεμαχισμό. Για μεγάλα όμως προγράμματα, που χρειάζονται πολύ χρόνο για να

εκτελεστούν, ο δυναμικός γράφος εξάρτησης εξαιτίας του ότι θα χρειαστεί αρκετό χώρο αποθήκευσης, θα χρειαστεί πολύ χρόνο να κτιστεί. Μάλιστα στις περισσότερες περιπτώσεις, η μνήμη δεν είναι αρκετή, αφού τα γραφήματα είναι πάρα πολύ μεγάλα, για αυτό το λόγο προτάθηκε ο αλγόριθμος καμίας επεξεργασίας (No Processing Algorithm), ο οποίος δεν εκτελεί καμία προεπεξεργασία.

Με βάση τον αλγόριθμο NP, οι δυναμικές εξαρτήσεις στοιχείων εξάγονται από το ίχνος εκτέλεσης βάσει της ζήτησης που υπάρχει κατά τη διάρκεια του χειρισμού των δυναμικών αιτημάτων τεμαχισμού. Επομένως κάθε φορά που εξετάζεται το ίχνος εκτέλεσης, μόνο οι εξαρτήσεις που είναι σχετικές με το τεμάχιο που υπολογίζεται εξάγονται από το ίχνος.

### 4.5.3 Αλγόριθμος Περιορισμένης Επεξεργασίας (Limited Processing Algorithm - LP)

Παρόλο που ο NP αλγόριθμός λύνει το πρόβλημα έλλειψης μνήμης που παρατηρείται στον FP αλγόριθμο, εντούτοις αντιμετωπίζει ένα άλλο πρόβλημα, το κόστος της αύξησης του χρόνου υπολογισμού του τεμαχίου. Ο χρόνος που χρειάζεται για να διασχίσουμε ένα μακρύ μονοπάτι εκτέλεσης είναι ένα σημαντικό μέρος του κόστους τεμαχισμού. Ενώ ο FP αλγόριθμος διασχίζει το μονοπάτι μόνο μια φορά για όλα τα αιτήματα τεμαχισμού, ο NP αλγόριθμός συχνά διασχίζει το ίδιο κομμάτι του μονοπατιού πολλές φορές, ανακαλύπτοντας κάθε φορά διαφορετικές σχετικές εξαρτήσεις για διαφορετικά αιτήματα τεμαχισμού. Οπότε μπορούμε να πούμε ότι ο NP αλγόριθμός κάνει λιγότερη προ-επεξεργασία, με αποτέλεσμα να αυξάνεται το κόστος τεμαχισμού, από την άλλη ο FP αλγόριθμος κάνοντας περισσότερη προ-επεξεργασία, δημιουργεί προβλήματα έλλειψης μνήμης.

Στον αντίποδα των πιο πάνω αλγορίθμων, βρίσκεται ο LP αλγόριθμος ο οποίος βρίσκει την ισορροπία μεταξύ προ-επεξεργασίας και κόστους τεμαχισμού. Αρχικά διεξάγει περιορισμένη προ-επεξεργασία του μονοπατιού εκτέλεσης, με σκοπό τον εμπλουτισμό του μονοπατιού με συνοπτική πληροφορία η οποία επιτρέπει την γρηγορότερη διάσχιση του επαυξημένου μονοπατιού. Στη συνέχεια χρησιμοποιείται ανάλυση με βάση τις

απαιτήσεις για τον υπολογισμό του τεμαχίου χρησιμοποιώντας το επαυξημένο μονοπάτι.

## 4.6 Αλγόριθμοι Δυναμικού Τεμαχισμού για Αντικειμενοστρεφή Προγράμματα

Έχουν γίνει αρκετές προτάσεις αλγορίθμων δυναμικού τεμαχισμού αναφορικά με αντικειμενοστρεφή προγράμματα, παρακάτω θα δούμε επιγραμματικά μερικές από αυτές. Γενικά, ο δυναμικός τεμαχισμός για ένα αντικειμενοστραφές πρόγραμμα εξαιτίας των ιδιοτήτων πολυμορφισμού και της δυναμικής σύνδεσης δεδομένων που παρέχει, διαφέρει από τα προγράμματα που δεν ανήκουν σε αυτή την κατηγορία, και για αυτούς ακριβώς τους λόγους, η διαδικασία του εντοπισμού εξαρτήσεων σε ένα αντικειμενοστραφές πρόγραμμα είναι περίπλοκη.

### 4.6.1 Αλγόριθμος Δυναμικού Τεμαχισμού για Αντικειμενοστρεφή Προγράμματα (Dynamic Object Oriented Dependence Graph - DOOG)

Ο Zhao [ZH98, , XZWC05] πρότεινε τη δημιουργία του δυναμικού αντικειμενοστραφούς γράφου εξάρτησης (Dynamic Object Oriented Dependence Graph - DOOG), ο οποίος περιέχει τις δυναμικές εξαρτήσεις για τα στιγμιότυπα δηλώσεων μίας συγκεκριμένης εκτέλεσης ενός αντικειμενοστραφούς προγράμματος. Σκοπός της υλοποίησης και εφαρμογής του εν λόγω αλγορίθμου είναι το «κτίσιμο» ενός δυνατού και αποδοτικού εργαλείου αποσφαλμάτωσης, το οποίο κατά τη διάρκεια ελέγχου του προγράμματος, χρησιμοποιεί την τεχνική τεμαχισμού για να φιλτράρει το πρόγραμμα. Ο δυναμικός αντικειμενοστραφής γράφος εξάρτησης (DOOG) είναι ένα γράφημα, έστω G(V,A), όπου, V ένα σύνολο κόμβων και A ένα σύνολο ακμών που αναπαριστούν τις εξαρτήσεις ελέγχου και τις εξαρτήσεις στοιχείων μεταξύ των κόμβων.

Για παράδειγμα, για το πρόγραμμα της Εικόνα 4.9 και για είσοδο `argv[1] = 3` παράγεται ο δυναμικός αντικειμενοστραφής γράφος εξάρτησης που φαίνεται παρακάτω (Εικόνα 4.10).

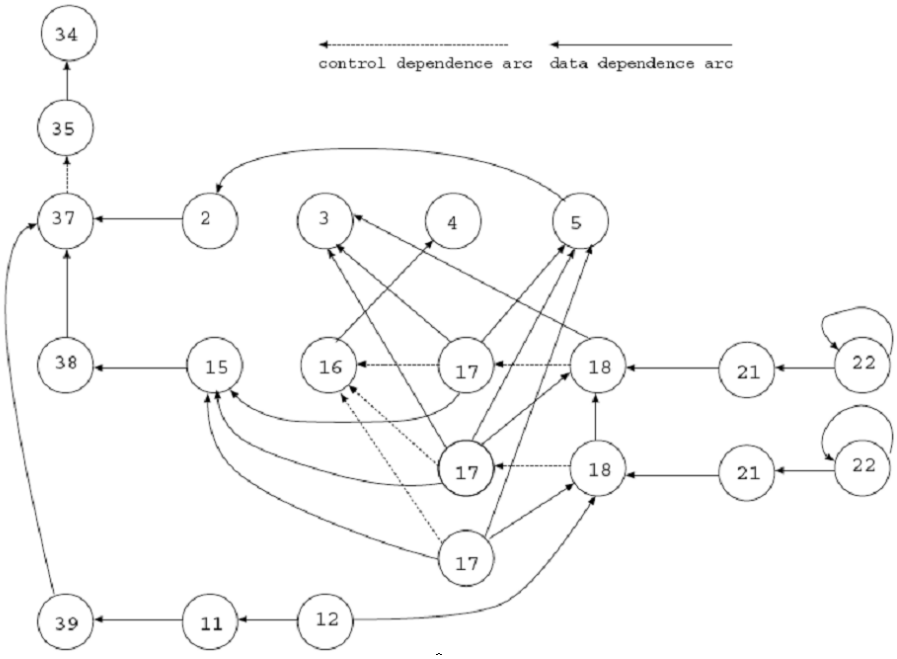
```
ce1:    class Elevator {
        public:
e2:       Elevator(int l_top_floor)
s3:         { current_floor = 1;
s4:           current_direction = UP;
s5:           top_floor = l_top_floor; }
e6:       virtual ~Elevator() { }
e7:       void up()
s8:         {current_direction = UP;}
e9:       void down()
s10:        { current_direction = DOWN; }
e11:      int which_floor()
s12:        { return current_floor; }
e13:      Direction direction()
s14:        { return current_direction;}

e15:      virtual void go(int floor)
s16:        { if (current_direction = UP)
s17:           { while (current_floor != floor
                  && (current_floor <= top_flo
c18:              add(current_floor, 1);}
              else
s19:           { while (current_floor != floor
                        && (current_floor > 0
c20:              add(current_floor, -1);}
            };
          private:
e21         add(int &a, const int &b)
s22:          { a = a + b; };
         protected:
           int  current_floor;
           Direction current_direction;
           int  top_floor;
         };

c23:     class AlarmElevator: public Elevator
         public:
e24:       AlarmElevator(int top_floor);
c25:         Elevator(top_floor)
s26:           { alarm_on = 0; }
e27:       void set_alarm()
s28:         { alarm_on = 1; }
e29:       void reset_alarm()
s30:         { alarm_on = 0; }
e31:       void go(int floor)
s32:         { if (!alarm_on)
c33:           Elevator::go(floor)
             };
           protected:
             int alarm_on;
           };

e34: main(int argc, char **argv) {
        Elevator *e_ptr;
s35:    if (argv[1])
s36:         e_ptr = new AlarmElevator(10);
        else
s37:         e_ptr = new Elevator(10);
c38:    e_ptr -> go(3);
s39:    cout << "\n Currently on floor:"
           << e_ptr -> which_floor() << "\n";
       }
```

**Εικόνα 4.11: Πρόγραμμα C++ [ZH98]**




**Εικόνα 4.10: Δυναμικός Αντικειμενοστραφής Γράφος Εξάρτησης [ZH98]**

Στη συνέχεια θα δούμε τα βασικά βήματα του αλγορίθμου για τον υπολογισμό ενός δυναμικού τεμαχίου ενός αντικειμενοστραφούς προγράμματος, χρησιμοποιώντας το γράφο DOOG.

- Βήμα 1ο: Υπολογισμός του δυναμικού τεμαχίου μέσω του γράφου DOOG του αντικειμενοστραφούς προγράμματος.
- Βήμα 2 α Συσχετισμός του τεμαχίου που υπολογίστηκε στο βήμα 1 με τον πηγαίο κώδικα, έτσι ώστε να πάρουμε το δυναμικό τεμάχιο του προγράμματος.

### 4.6.2 Άλλοι αλγόριθμοι δυναμικού τεμαχισμού για Αντικειμενοστρεφή Προγράμματα

Την πρόταση του Zhao ακολούθησαν οι Song και Huynh [SH99, XZWC05] οι οποίοι ανέλυσαν τις παραμέτρους αντικειμενοστραφών προγραμμάτων και παρουσίασαν την έννοια των Δυναμικών Διαγραμμάτων Συσχετίσεων μεταξύ των Αντικειμένων (Dynamic Object Relation Diagrams - DORD).

Έπειτα, οι Xu, Chen και Yang [XCY02, XZWC05] πρότειναν έναν αλγόριθμο δυναμικού τεμαχισμού για αντικειμενοστραφή προγράμματα, που είναι βασισμένος

στην ανάλυση εξαρτήσεων. Αυτή η προσέγγιση χρησιμοποίησε τον Object Program Dependence Graph (OPDG) και άλλες στατικές πληροφορίες, με σκοπό να μειώσει τις πληροφορίες που επισημαίνονται κατά τη διάρκεια της εκτέλεσης προγράμματος. Ο αλγόριθμος αυτός αποτελεί ένα συνδυασμό της προς τα εμπρός (forward) και προς τα πίσω (backward) ανάλυσης.

Πολλές φορές αντικείμενα περνιούνται σαν παράμετροι σε μεθόδους, όμως οι περισσότεροι αλγόριθμοι που προτάθηκαν δεν μπορούν να διαχειριστούν σωστά αυτές τις περιπτώσεις μιας και κρίνονται ανεπαρκείς στη διαχείριση επαναλαμβανόμενων (recursive) δομών δεδομένων. Αυτό έχει σαν επακόλουθο τα τεμάχια που παράγονται να είναι είτε λανθασμένες (πολύ μικρές), είτε ανακριβής (πολύ μεγάλες). Οι Hammer και Snelting [HS04], παρουσίασαν ένα καινούργιο αλγόριθμο για δημιουργία φετών προγράμματος (program slices) Java προγραμμάτων που διαχειρίζεται περιπτώσεις όπου αντικείμενα περνιούνται σαν παράμετροι σε μεθόδους, κάνοντας χρήση System Dependence Graph (SDG). Τέλος συνέκριναν τον αλγόριθμο το φ με τον ανταγωνιστικό αλγόριθμό που παρουσιάζεται στο [LH98] και κατέληξαν στο συμπέρασμα ότι ο δικός τους αλγόριθμος είναι πιο ακριβής.

## 4.7 Αλγόριθμοι Δυναμικού Τεμαχισμού για εντοπισμό της Ελαττωματικής περιοχής (Dynamic Slicing for Fault Location)

Παρακάτω θα περιγραφούν αλγόριθμοι Δυναμικού Τεμαχισμού που στοχεύουν στη συλλογή λανθασμένων δηλώσεων στα προγράμματα. Από την εφαρμογή των αλγορίθμων αυτών παράγονται δύο είδη δυναμικών φετών (dynamic slices):

- Με κατεύθυνση προς τα πίσω (Backward)
- Με κατεύθυνση προς τα εμπρός (Forward)

Ένα δυναμικό τεμάχιο προγράμματος με κατεύθυνση προς τα πίσω (Backward Dynamic Slice) μιας μεταβλητής σε ένα σημείο του μονοπατιού εκτέλεσης περιλαμβάνει όλες εκείνες τις δηλώσεις του προγράμματος που επηρεάζουν την τιμή της μεταβλητής σε εκείνο το σημείο.

Αντίθετα, ένα δυναμικό τεμάχιο προγράμματος με κατεύθυνση προς τα εμπρός (Forward Dynamic Slice) μιας μεταβλητής σε ένα σημείο του μονοπατιού εκτέλεσης περιλαμβάνει όλες εκείνες τις δηλώσεις του προγράμματος που επηρεάστηκαν από την τιμή της μεταβλητής σε εκείνο το σημείο.

Σημειώνεται ότι όταν αναφερόμαστε σε Δυναμικό Τεμαχισμό χωρίς να γίνεται διαχωρισμός κατεύθυνσης, εννοούμε με κατεύθυνση προς τα πίσω. Το ίδιο ισχύει και για την εργασία αυτή.

Έχουν προταθεί δύο τύποι μεθόδων για τον υπολογισμό των δυναμικών φετών με κατεύθυνση προς τα πίσω (backward dynamic slices), οι προς τα πίσω μέθοδοι υπολογισμού (backward computation methods) και οι προς τα εμπρός μέθοδοι υπολογισμού (forward computation methods). Στις προς τα πίσω μεθόδους υπολογισμού, οι εξαρτήσεις του προγράμματος που παράγονται κατά τη διάρκεια μιας εκτέλεσής του, αποθηκεύονται στην μορφή ενός δυναμικού εξαρτώμενου γράφου (dynamic dependence graph). Στις προς τα εμπρός μεθόδους υπολογισμού, οι τελευταίες προς τα πίσω δυναμικά τεμάχια όλων των μεταβλητών προγράμματος υπολογίζονται και διατηρούνται ως σύνολα δηλώσεων καθώς το πρόγραμμα εκτελείται. Το πλεονέκτημα αυτής της προσέγγισης είναι ότι το χωρικό κόστος δεν είναι πλέον ανάλογο προς το μήκος της εκτέλεσης αλλά ως προς τον αριθμό των μεταβλητών.

Παρακάτω παρουσιάζουμε τρεις δυναμικούς αλγορίθμους τεμαχισμού:

- Αλγόριθμος Τεμαχισμού Στοιχείων (Data Slicing)
- Αλγόριθμος Ολοκληρωμένου Τεμαχισμού (Full Slicing)
- Αλγόριθμος Σχετικού Τεμαχισμού (Relevant Slicing).

### 4.7.1 Αλγόριθμος Τεμαχισμού Στοιχείων (Data Slicing)

Στον αλγόριθμο Τεμαχισμού Στοιχείων (Data Slicing) [ZHGG05, ZGG06] παράγονται δυναμικές εξαρτήσεις στοιχείων (dynamic data dependences), οι οποίες συμπεριλαμβάνονται μέσα στα τεμάχια προγράμματος (data slices) και αφορούν τις δηλώσεις του προγράμματος που άμεσα ή έμμεσα συντελούν στον υπολογισμό

λανθασμένης εξόδου. Στην συνέχεια δίνουμε μια σύντομη περιγραφή του προαναφερθέντος αλγορίθμου.

Δεδομένης μίας δήλωσης s, ορίζουμε ως $s_i$ το $i^{στό}$ στιγμιότυπο εκτέλεσης του s. Επίσης, ορίζουμε ως $Def[s_i]$ το σύνολο των μεταβλητών που καθορίζονται από το $s_i$ και ως $Use[s_i]$ τα σύνολα των μεταβλητών που διαβάζονται από το $s_i$. Το $s_i$ είναι δυναμικά εξαρτώμενο στοιχείο (dynamically data dependent) από ένα άλλο στιγμιότυπο εκτέλεσης $m_n$ , εάν και μόνο εάν υπάρχει μία μεταβλητή, έστω u, έτσι ώστε $u \in Def[m_n]$ και $u \in Use[s_i]$.

Παρακάτω παρουσιάζουμε την προς τα εμπρός (forward) μέθοδο υπολογισμού δυναμικών φετών (χρησιμοποιείται ο συμβολισμός DS[u] για τον πιο πρόσφατη δήλωση του u). Να σημειωθεί, ότι ο αλγόριθμος υπολογίζει τα δυναμικά τεμάχια καθώς εκτελούνται οι δηλώσεις. Επίσης, αν και υπολογίζονται όλα τα τεμάχια, αποθηκεύονται μόνο τα πιο πρόσφατα τεμάχια όλων των μεταβλητών. Αφού ολοκληρωθεί το $s_i$, το $Def[s_i]$ θα πρέπει να ενημερωθεί, έτσι ώστε να συμπεριλάβει τις δηλώσεις που ανήκουν στα πιο πρόσφατα δυναμικά τεμάχια των μεταβλητών που χρησιμοποιούνται από το $s_i$ (δηλ., μεταβλητές που ανήκουν στο $Use[s_i]$) και την ίδια τη δήλωση s. Ο αλγόριθμος ενημέρωσης των δυναμικών φετών μετά από την εκτέλεση του $s_i$ φαίνεται στην Εικόνα 4.11.

**Algorithm 1** *Updating Data Slicing Information*

**Procedure** Update($s_i$)

1: $slice = \{\}$;
2: **for** (each use $v$ in $Use[s_i]$) **do**
3: $slice = slice \cup DS[v]$;
4: **end for**
5: **for** (each definition $v$ in $Def[s_i]$) **do**
6: $DS[v] = slice \cup \{s\}$;
7: **end for**

**Εικόνα 4.12: Αλγόριθμος Τεμαχισμού Στοιχείων (Data Slicing) [ZHGG05]**

### 4.7.2 Αλγόριθμος Ολοκληρωμένου Τεμαχισμού (Full Slicing)

Στον αλγόριθμο Ολοκληρωμένου Τεμαχισμού [ZHGG05, ZGG06] παράγονται δυναμικές εξαρτήσεις στοιχείων (dynamic data dependences) και δυναμικές εξαρτήσεις ελέγχου, οι οποίες συμπεριλαμβάνονται μέσα στα ολοκληρωμένα τεμάχια (full slices) και αφορούν τις δηλώσεις του προγράμματος και που άμεσα ή έμμεσα συντελούν στον υπολογισμό λανθασμένης εξόδου.

Ο αλγόριθμος Ολοκληρωμένου Τεμαχισμού διαφοροποιείται από τον αλγόριθμο τεμαχισμού στοιχείων ως προς την εισαγωγή εξαρτήσεων ελέγχου (control dependencies). Για αυτόν ακριβώς το λόγο, για το δεύτερο λάθος που φαίνεται στον πίνακα της Εικόνα 4.14 το τεμάχιο στοιχείων (data slice) αποτυγχάνει να συμπεριλάβει την λανθασμένη δήλωση της γραμμής 10, ενώ το ολοκληρωμένο τεμάχιο (full slice) χειρίζεται το παραπάνω με επιτυχία.

Μια δήλωση εξάρτησης ελέγχου s ως προς ένα κατηγόρημα έστω p, είναι αληθής ή λανθασμένη, εάν και μόνο εάν το αποτέλεσμα του p (true / false) καθορίσει εάν το s θα εκτελεσθεί. Τα ολοκληρωμένα τεμάχια (full slices), υπολογίζονται διασχίζοντας τις ακμές με βάση τις εξαρτήσεις στοιχείων και ελέγχου αρχίζοντας από την τιμή εξόδου.

**Algorithm 2** *Updating Full Slicing Information*

**Procedure** Update($s_i$, $stack$)
1: $slice = \{\}$;
2: **for** (each use $v$ in $Use[s_i]$) **do**
3: $slice = slice \cup FS[v]$;
4: **end for**
5: $cd$ = the predicate in CD(s) s.t. $stack.ts[cd]$ is maximum;
6: $slice = slice \cup stack.slice[cd] \cup \{cd\}$;
7: **if** ($s$ is a predicate) **then**
8: $stack.slice[s] = slice$;
9: $stack.ts[s]$=$timestamp$++;
10: **end if**
11: **for** (each definition $v$ in $Def[s_i]$) **do**
12: $FS[v] = slice \cup \{s\}$;
13: **end for**

**Εικόνα 4.13: Αλγόριθμος Ολοκληρωμένου Τεμαχισμού (Full Slicing) [ZHGG05]**

Στην Εικόνα 4.12 παρουσιάζουμε τον προς τα εμπρός αλγόριθμο για υπολογισμό ολοκληρωμένων φετών. Με `FS[u]` συμβολίζεται το ολοκληρωμένο τεμάχιο για την τελευταία δήλωση της μεταβλητής `u`. Η μεταβλητή `timestamp` δείχνει τον τρέχοντα χρόνο, ενώ με `stack` συμβολίζεται η στοίβα. Τα `stack.slice[]` και `stack.ts[]` είναι πίνακες που διατίθενται στο τρέχον πλαίσιο της στοίβας. Για να υπολογιστούν σωστά οι εξαρτήσεις στην περίπτωση των επαναλαμβανόμενων κλήσεων, αποθηκεύουμε timestamps των πιο πρόσφατων εκτελέσεων των κατηγορημάτων και των αντίστοιχων πλήρων φετών τους στα `stack.ts[]` και `stack.slice[]` αντίστοιχα. Αυτό εγγυάται ότι, όποτε ψάχνουμε για την περίπτωση κατηγορήματος με το μεγαλύτερο `timestamp` στο `CD(ι)`, εξετάζουμε μόνο εκείνους που έχουν το ίδιο πλαίσιο στοίβας με το `Si`. Παρατηρούμε ότι η δομή αυτού του αλγορίθμου είναι παρόμοια με τον αλγόριθμο τεμαχισμού στοιχείων (data slicing). Οι επιπρόσθετες δηλώσεις επιτρέπουν το χειρισμό του εξαρτήσεων ελέγχου κατά τον τεμαχισμό.

### 4.7.3 Αλγόριθμος Σχετικού Τεμαχισμού (Relevant Slicing)

Ο αλγόριθμος Σχετικού Τεμαχισμού [ZHGG05, ZGG06] αποτελεί προέκταση των αλγορίθμων τεμαχισμού στοιχείων και ολοκληρωμένου τεμαχισμού αντίστοιχα που παρουσιάστηκαν παραπάνω.

Τα σχετικά τεμάχια εκτός από τις εξαρτήσεις στοιχείων (data) και ελέγχου (control dependences) περιλαμβάνουν και κατηγορήματα που στην πραγματικότητα δεν έχουν επηρεάσει το αποτέλεσμα, αλλά θα μπορούσαν να το είχαν επηρεάσει εάν είχαν υπολογιστεί διαφορετικά.

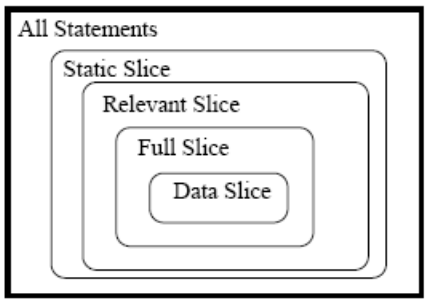
Ας πάρουμε το παράδειγμα της Εικόνας 4.14, στην περίπτωση του σφάλματος Error3 (τελευταία στήλη του πίνακα), η εσφαλμένη δήλωση 7 δεν περιέχεται στο Full Slice. Το σφάλμα αυτό έχει σαν αποτέλεσμα η μεταβλητή α στη $7_1$ να πάρει τιμή 1, αντί 4, με αποτέλεσμα το κατηγόρημα της δήλωσης $9_1$ να αποτιμηθεί διαφορετικά από ότι στην ορθή εκτέλεση του προγράμματος, αλλάζοντας τη ροή εκτέλεσης και η δήλωση 10 να

μην εκτελεστεί, πράγμα που θα συνέβαινε στην ορθή εκτέλεση του προγράμματος. Λόγω παράληψης εκτέλεσης της δήλωσης 10, η $9_1$ δεν συμπεριλαμβάνεται στο slice και κατ' επέκταση ούτε και η $7_1$ δήλωση περιλαμβάνεται στο Full slice. Γενικά αυτό συμβαίνει όταν μερικές δηλώσεις που θα έπρεπε να είχαν εκτελεστεί, δεν εκτελέστηκαν λόγω του λάθους.

### 4.7.4 Γενικά Σχόλια για αλγορίθμους Δυναμικού Τεμαχισμού για Εντοπισμό Λαθών

Από δοκιμές των παραπάνω αλγορίθμων, έχει παρατηρηθεί ότι η αποδοτικότητά τους καθορίζεται κυρίως από δύο παράγοντες: πόσο συχνά οι ελαττωματικές δηλώσεις κώδικα βρίσκονται μέσα στο δυναμικό τεμάχια και κατά δεύτερο, το μέγεθος του τεμαχίου, δηλαδή πόσες δηλώσεις περιλαμβάνονται σε αυτό.

Όπως φαίνεται και στην Εικόνα 4.13 [ZHGG05, ZGG06] μεταξύ των δυναμικών φετών επικρατεί η σχέση: Στατικό Τεμάχιο $\supseteq$ Σχετικό Τεμάχιο $\supseteq$ Ολοκληρωμένο Τεμάχιο $\supseteq$ Τεμάχιο Στοιχείων ως προς τον αριθμό των ελαττωματικών δηλώσεων που μπορούν να αιχμαλωτιστούν σε κάθε μία από τα τεμάχια στατικού και δυναμικού τεμαχισμού αντίστοιχα. Για παράδειγμα, εάν ο προγραμματιστής έχει κάνει λάθος που σχετίζεται με δήλωση ανάθεσης, η λανθασμένη δήλωση είναι σχεδόν σίγουρο ότι θα περιέχεται στο σχετικό τεμάχιο, ενώ ίσως να μην περιέχεται στο ολοκληρωμένο τεμάχιο και στο τεμάχιο στοιχείων.




**Εικόνα 4.14: Σχέση μεταξύ αλγορίθμων δυναμικού τεμαχισμού για Εντοπισμό Λαθών (Fault Location) [ZHGG05, ZGG06]**

Για το πρόγραμμα που φαίνεται στην αριστερή στήλη του πίνακα (Εικόνα 4.14), δίνονται τα αποτελέσματα των αλγορίθμων και αντίστοιχα η παραγωγή του τεμαχίου

στοιχείων, του ολοκληρωμένου τεμαχίου και του σχετικού τεμαχίου με τρία διαφορετικά λάθη στο πρόγραμμα. Στην περίπτωση του πρώτου λάθους, η λανθασμένη δήλωση, μπορεί να βρεθεί σε όλα τα τεμάχια και επομένως στην περίπτωση αυτή θα επιλέγαμε τον αλγόριθμο Τεμαχισμού Στοιχείων (Data Slicing), αφού το μέγεθος του τεμαχίου που παράγει είναι το πιο μικρό. Σε περίπτωση του δεύτερου λάθους, το τεμάχιο στοιχείων, δεν περιέχει την ελαττωματική δήλωση (10), αλλά μπορεί να βρεθεί με Ολοκληρωμένο Τεμαχισμό (FS-Full Slicing) Τέλος, στην περίπτωση του τρίτου λάθους, η ελαττωματική δήλωση (7) μπορεί μόνο να βρεθεί μόνο με Σχετικό Τεμαχισμό (RS- Relevant Slicing). Σημειώνεται ότι το σύνολο ES αντιπροσωπεύει το σύνολο των δηλώσεων που έχουν εκτελεστεί με βάση τις τιμές εισόδου.

| | | | |
|---|---|---|---|
| 1. read (a);<br>2. read (n);<br>3. i=0;<br>4. while (i<n) {<br>5. read (x);<br>6. read (y);<br>7. a=a/x;<br>8. b=x;<br>9. if (a>1)<br>10. b=a-4;<br>11. if (b>0)<br>12. z=x+y;<br>else<br>13. z=x-y;<br>14. output (z);<br>15. i=i+1;<br>} | $(Error1)$<br>13. $z = x - y$<br>$\rightarrow$ 13. $z = x - y + 1$<br>Input: $a = 2;\ n = 1;$<br>$x = -1;\ y = 1;$<br>Wrong output: $z = -1;$<br>Correct output: $z = -2;$<br>$*DS = \{5, 6, 13\}$<br>$FS = \{1, 2, 3, 4, 5, 6, 8, 11, 13\}$<br>$RS = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13\}$<br>$ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13, 14, 15\}$ | $(Error2)$<br>10. $b = a - 4$<br>$\rightarrow$ 10. $b = a - 3$<br>Input: $a = 8;\ n = 1;$<br>$x = 2;\ y = 2;$<br>Wrong output: $z = 4;$<br>Correct output: $z = 0;$<br>$DS = \{5, 6, 12\}$<br>$*FS = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$<br>$RS = \{1, 2, 3, 4, 5, 6, 7, 9, 10, 11, 12\}$<br>$ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 14, 15\}$ | $(Error3)$<br>7. $a = a/x$<br>$\rightarrow$ 7. $a = a/2x - 1;$<br>Input: $a = 8;\ n = 1;$<br>$x = 2;\ y = 2;$<br>Wrong output: $z = 4;$<br>Correct output: $z = 0;$<br>$DS = \{5, 6, 12\}$<br>$FS = \{1, 2, 3, 4, 5, 6, 8, 11, 12\}$<br>$*RS = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12\}$<br>$ES = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 14, 15\}$ |

**Εικόνα 4.15: Παραδείγματα φετών που παράγονται από Data Slicing, Full Slicing και Relevant Slicing [ZHGG05]**

Από το παραπάνω παράδειγμα συμπεραίνουμε ότι τα διαφορετικά δυναμικά τεμάχια που εξετάζουμε διαφέρουν στο μέγεθος και τη δυνατότητά τους να συλλάβουν τις ελαττωματικές δηλώσεις.

Επίσης, έχει παρατηρηθεί ότι τα τεμάχια που παράγονται με τη βοήθεια των αλγορίθμων Ολοκληρωμένου τεμαχισμού και Σχετικού τεμαχισμού αντίστοιχα, μειώνουν σημαντικά το μέγεθος του προγράμματος που πρέπει να εξεταστεί με στόχο τον εντοπισμό λανθασμένων δηλώσεων.

# Κεφάλαιο 5

# Γενετικοί Αλγόριθμοι

---



---

## 5.1 Εισαγωγή στους Γενετικούς Αλγόριθμους

Γενετικός Αλγόριθμος (Genetic Algorithm-GA) είναι ένας αλγόριθμος αναζήτησης ο οποίος βασίζεται στις αρχές της φυσικής επιλογής και της γενετικής αναπαραγωγής [HO75, GO89]. Οι Γ.Α. χρησιμοποιούν ορολογία δανεισμένη από τον χώρο της Φυσικής Γενετικής. Αναφέρονται σε άτομα (individuals) ή γενότυπους (genotypes) μέσα σε ένα πληθυσμό. Κάθε άτομο ή γενότυπος αποτελείται από χρωματοσώματα (chromosomes). Στους Γ.Α. αναφερόμαστε συνήθως σε ένα μόνο χρωματόσωμα. Τα χρωματοσώματα αποτελούνται από γονίδια (gens) που είναι διατεταγμένα σε μία γραμμική ακολουθία. Κάθε γενότυπος αναπαριστά μία πιθανή λύση σε ένα πρόβλημα. Μια διαδικασία εξέλιξης που εφαρμόζεται πάνω σε ένα πληθυσμό αντιστοιχεί σε ένα εκτενές ψάξιμο στο χώρο των πιθανών λύσεων.

Οι Γ.Α. διατηρούν ένα πληθυσμό πιθανών λύσεων, του προβλήματος που μας ενδιαφέρει, πάνω στον οποίο δουλεύουν, σε αντίθεση με άλλες μεθόδους αναζήτησης που επεξεργάζονται ένα μόνο σημείο του διαστήματος αναζήτησης. Έτσι ένας Γ.Α. πραγματοποιεί αναζήτηση σε πολλές κατευθύνσεις και υποστηρίζει καταγραφή και ανταλλαγή πληροφοριών μεταξύ αυτών των κατευθύνσεων. Ο πληθυσμός υφίσταται μια προσομοιωμένη γενετική εξέλιξη. Σε κάθε γενιά οι «σχετικά» καλές λύσεις αναπαράγονται ενώ οι σχετικά «κακές» απομακρύνονται. Ο διαχωρισμός και η αποτίμηση των διαφόρων λύσεων γίνεται με τη βοήθεια μια συνάρτησης καταλληλότητας (fitness function), η οποία παίζει το ρόλο του περιβάλλοντος μέσα στο οποίο εξελίσσεται ο πληθυσμός.

Τα κύρια χαρακτηριστικά που διαχωρίζουν τους Γ.Α. από άλλες μεθόδους αναζήτησης είναι:

- Ο πληθυσμός από άτομα (population of individuals), όπου κάθε άτομο παριστάνει μία πιθανή λύση του προβλήματος.
- Η συνάρτηση καταλληλότητας (fitness function) η οποία αξιολογεί τη χρησιμότητα κάθε ατόμου ως λύση του προβλήματος.
- Η συνάρτηση επιλογής (selection function) η οποία επιλέγει άτομα (γονείς) για αναπαραγωγή με βάση την τιμή της συνάρτησης καταλληλότητας.
- Οι γενετικοί τελεστές (genetic operators) οι οποίοι αλλάζουν συγκεκριμένα άτομα για την παραγωγή νέων ατόμων. Αυτοί οι τελεστές για παράδειγμα η διασταύρωση (crossover) και η μετάλλαξη (mutation), προσπαθούν να καλύψουν το εύρος αναζήτησης. Με την διασταύρωση τα μέλη του πληθυσμού που έχουν επιλεγεί ανταλλάζουν γενετικό υλικό, ενώ ο τελεστής της μετάλλαξης προκαλεί μικρές αλλαγές στην τιμή ενός ή περισσότερων μεταβλητών του μέλους ή του γονιδίου του πληθυσμού.

Επιπρόσθετα ένα άλλο βασικό στοιχείο των Γ.Α. όπως και κάθε αλγόριθμου είναι το κριτήριο τερματισμού. Το κριτήριο αυτό θα πρέπει κατά πρώτον να μην επιτρέπει την εκτέλεση επαναλήψεων που δεν οδηγούν σε κάποιο αποτέλεσμα και τα δεύτερον να διακόπτει τον αλγόριθμο όταν με κάποια βεβαιότητα έχει βρεθεί ο στόχος.

Παρακάτω παραθέτουμε τα κύρια βήματα ενός τυπικού Γ.Α.:

1. Αρχικοποίησε τον πληθυσμό με τυχαία αναπαραγωγή χρωματοσωμάτων.
2. Επανέλαβε μέχρι να ικανοποιηθεί η συνθήκη τερματισμού
   Αξιολόγησε τον τρέχον πληθυσμό χρησιμοποιώντας τη συνάρτηση καταλληλότητας.
   Επέλεξε τους γονείς
   Εφάρμοσε γενετικούς τελεστές στους γονείς για να αναπαραχθούν τα παιδιά
3. Θέσε τον τρέχον πληθυσμό ίσο με τον πληθυσμό των παιδιών

Μερικά από τα πλεονεκτήματα των Γ.Α. είναι ότι μπορούν να επιλύσουν δύσκολα προβλήματα γρήγορα και αξιόπιστα, μπορούν εύκολα να συνεργαστούν με τα υπάρχοντα μοντέλα και συστήματα, είναι εύκολα επεκτάσιμοι και εξελίξιμοι, μπορούν να συμμετέχουν σε υβριδικές μορφές με άλλες μεθόδους, εφαρμόζονται σε πολύ περισσότερα πεδία από κάθε άλλη μέθοδο, δεν απαιτούν περιορισμούς στις συναρτήσεις που εφαρμόζονται, δεν ενδιαφέρει η σημασία της υπό εξέταση πληροφορίας, έχουν από τη φύση τους το στοιχείο του παραλληλισμού, είναι μια μέθοδος που κάνει ταυτόχρονα εξερεύνηση του χώρου αναζήτησης και εκμετάλλευση της ήδη επεξεργασμένης πληροφορίας και τέλος επιδέχονται παράλληλη υλοποίηση.

Ωστόσο όπως κάθε άλλη μέθοδος έχουν και μειονεκτήματα τα κυριότερα είναι ότι σε πολλές περιπτώσεις έχουν αργό χρόνο σύγκλισης και σε αρκετές περιπτώσεις τείνουν να εγκλωβιστούν σε τοπικά ελάχιστα της συνάρτησης καταλληλότητας. Τα μειονεκτήματα όμως που έχουν αναφερθεί μπορούν να αντιμετωπιστούν με προσεκτικό σχεδιασμό του Γ.Α. και με κατάλληλη προσαρμογή των παραμέτρων του στο πρόβλημα προς επίλυση.

Οι Γ.Α. βρίσκουν εφαρμογή σχεδόν σε κάθε πρόβλημα Τεχνητής Νοημοσύνης εξαιτίας της απλότητας τους και των καλών αποτελεσμάτων που επιτυγχάνουν. Μερικές από τις πολλές εφαρμογές τους είναι η δημιουργία γραφικών σε υπολογιστή, η σύνθεση μουσικής, η κατασκευή τεχνιτών νευρωνικών δικτύων, η επεξεργασία εικόνων, η μηχανική μάθηση, η εύρεση μέγιστης τιμής αριθμητικών συναρτήσεων, η συνδυαστική βελτιστοποίηση στην οποία συγκαταλέγονται μεταξύ άλλων το πρόβλημα του πλανόδιου πωλητή, της αποθήκευσης κιβωτίων, της σχεδίασης VLSI κυκλωμάτων, του καταμερισμού εργασιών και του ωρολογίου προγράμματος.

Παράδειγμα εφαρμογής Γ.Α.

Για να κατανοήσουμε καλύτερα την λογική των Γ.Α. ας δούμε ένα μικρό παράδειγμα εφαρμογής ενός Γ.Α. Το πρόβλημα ορίζεται ως εξής με βάση τα ψηφία `0` έως `9` και τους τελεστές `+`, `-`, `*` και `/`, βρες μια ακολουθία που θα αντιπροσωπεύει ένα συγκεκριμένο αριθμό. Οι τελεστές θα εφαρμόζονται σειριακά ξεκινώντας από αριστερά προς δεξιά. Έτσι αν μας δοθεί ο αριθμός `23`, η ακολουθία `6+5*4/2+1` αποτελεί μια

πιθανή λύση του προβλήματος. Εάν ο αριθμός `75.5` είναι ο αριθμός που έχει επιλεγεί τότε η ακολουθία `5/2+9*7-5` αποτελεί μια πιθανή λύση.

Προχωρούμε με την Κωδικοποίηση του πληθυσμού, πρέπει να κωδικοποιήσουμε κάθε πιθανή λύση ως μια στοιχειοσειρά από ψηφία (bits). Πρέπει πρώτα από όλα να αναπαραστήσουμε όλους τους διαφορετικούς χαρακτήρες, δηλαδή τους αριθμούς `0` έως `9` και τους τελεστές `+`, `-`, `*` και `/`. Αυτή αναπαράσταση θα απεικονίζει ένα γονίδιο, κάθε χρωματόσωμα αποτελείτε από πολλά γονίδια. Για την αναπαράσταση των χαρακτήρων που χρησιμοποιούνται χρειάζονται τέσσαρα ψηφία (bits). Παρακάτω βλέπουμε όλα τα διαφορετικά γονίδια που χρειάζονται για την κωδικοποίηση του προβλήματος όπως έχει περιγραφεί:

0: 0000
1: 0001
2: 0010
3: 0011
4: 0100
5: 0101
6: 0110
7: 0111
8: 1000
9: 1001
+: 1010
-: 1011
*: 1100
/: 1101

Παρατηρούμε ότι τα πιθανά γονίδια `1110` και `1111` δεν χρησιμοποιούνται και θα αγνοηθούν από τον αλγόριθμο στην περίπτωση που παραχθούν.

Έτσι λοιπόν η λύση που έχει αναφερθεί πιο πάνω για τον αριθμό `23`, `' 6+5*4/2+1'` θα αναπαρασταθεί από εννέα γονίδια όπως φαίνεται πιο κάτω:

```
0110 1010 0101 1100 0100 1101 0010 1010 0001
6    +    5    *    4    /    2    +    1
```

Αυτά τα γονίδια στοιχειοθετούνται μαζί για να σχηματίσουν το χρωματόσωμα:

```
011010100101110001001101001010100001
```

Όσον αφορά την αποκωδικοποίηση, επειδή ο αλγόριθμος ασχολείται με τυχαίες αναπαραστάσεις ψηφίων, συχνά θα πρέπει να αποκωδικοποιηθεί κάποια σειρά από ψηφία όπως: `00100010101011101011101110010`

Η αποκωδικοποίηση των εν λόγω ψηφίων δίνει την πιο κάτω σειρά:

```
0010 0010 1010 1110 1011 0111 0010
2    2    +    n/a  -    7    2
```

Η σειρά αυτή δεν έχει νόημα για το εν λόγω πρόβλημα. Έτσι κατά την αποκωδικοποίηση ο αλγόριθμος θα αγνοεί όποια γονίδια δεν συμμορφώνονται με την με την πιο κάτω μορφή: αριθμός -> τελεστής -> αριθμός -> τελεστής …κτλ.

Η απόφαση για την συνάρτηση καταλληλότητας μπορεί να θεωρηθεί το πιο δύσκολο κομμάτι ενός Γ.Α.. Εξαρτάται πάντα από το πρόβλημα που έχουμε να λύσουμε αλλά η γενική ιδέα είναι ότι όσο πιο κοντά ένα χρωματόσωμα είναι προς τη λύση, τόσο πιο ψηλή βαθμολογία θα του δίνεται. Όσον αφορά το παράδειγμα μας, η συνάρτηση καταλληλότητας μπορεί να είναι αντιστρόφως ανάλογη της διαφοράς μεταξύ της λύσης και της τιμής που αντιπροσωπεύει το αποκωδικοποιημένο χρωματόσωμα.

Αν υποθέσουμε ότι ο επιθυμητός αριθμός για το παράδειγμα μας είναι `42`, για το χρωματόσωμα `011010100101110001001101001010100001` η συνάρτηση καταλληλότητας έχει τιμή `1/(42-23)` ή `1/19`.

## 5.2 Χρήση γενετικών αλγόριθμων για τεμαχισμό προγράμματος και εντοπισμό σφάλματος

Όπως αναφέρθηκε και πιο πάνω οι Γ.Α. έχουν εξεταστεί κατά πολύ στη βιβλιογραφία έχοντας ένα ευρύ φάσμα εφαρμογών. Το ίδιο ισχύει και για τους αλγορίθμους τεμαχισμού. Ωστόσο έχει γίνει ελάχιστη δουλεία που να αφορά την εφαρμογή γενετικών αλγορίθμων στη μηχανική λογισμικού (software engineering), ενώ δεν έχει βρεθεί στην βιβλιογραφία καμία εφαρμογή γενετικών αλγορίθμων για δημιουργία φετών προγράμματος (program slice construction) για εντοπισμό σφάλματος.

Αν και η δουλεία των Tao Jiang, Nicolas Gold, Mark Harman and Zheng Li στο [JGHL08] δεν σχετίζεται με τεμαχισμό προγράμματος για εντοπισμό σφάλματος θα ήταν καλό να γίνει μια αναφορά στην προσέγγιση που εισήγαγαν μιας και αποτέλεσε πηγή έμπνευσης για την παρούσα εργασία ο τρόπος που κωδικοποιούν το πληθυσμό του Γ.Α.. Στο [JGHL08] παρουσιάστηκε μια προσέγγιση για εντοπισμό δομών εξαρτήσεων (dependence structures) σε ένα πρόγραμμα με αναζήτηση στο υπερσύνολο του συνόλου όλων των πιθανών φετών προγράμματος, με σκοπό τον εντοπισμό εκείνου του συνόλου φετών προγράμματος οι οποίες αποσυνθέτουν ένα πρόγραμμα με τέτοιο τρόπο ώστε να αυξάνεται η κάλυψη προγράμματος και να ελαττώνεται η κάλυψη μεταξύ των φετών προγράμματος. Το πρόβλημα αυτό διατυπώνεται σαν πρόβλημα αναζήτησης μηχανικής λογισμικού.

Για την μοντελοποίηση του προβλήματος σαν πρόβλημα γενετικού αλγορίθμου καθορίζονται οι πιο κάτω παράμετροι:

- Χώρος αναζήτησης: είναι το σύνολο όλων των δυνατών φετών προγράμματος. Ένα πιθανό κριτήριο τεμαχισμού είναι τέτοιο ώστε να σημαίνει «κάθε κόμβος του System Dependence Graph (SDG) του προγράμματος». Επομένως για ένα πρόγραμμα με n κόμβους στο SDG, αντιστοιχούν n κριτήρια τεμαχισμού, άρα $2^n$ υποσύνολα κριτηρίων τεμαχισμού τα οποία αποτελούν και τον χώρο αναζήτησης του προβλήματος. Είναι εμφανές ότι αφού το n μπορεί να αυθαίρετα μεγάλο, η

απαρίθμηση δεν είναι εφικτή για αυτό το λόγο κρίνεται εύλογη η ανάγκη για χρήση γενετικού αλγορίθμου.

- Κωδικοποίηση του τεμαχισμού: Η κωδικοποίηση των πιθανών λύσεων είναι δυαδική. Ο ορισμός της αναπαράστασης του τεμαχισμού μπορεί να διατυπωθεί σαν ένας απλός πίνακας δύο διαστάσεων: ας υποθέσουμε ότι το `A[i; j]` είναι ένα δυαδικό ψηφίο, όπου `i` είναι ένα σημείο του προγράμματος και `j` ένα κριτήριο τεμαχισμού, έτσι ώστε `A[i; j] = 1` στην περίπτωση που το τεμάχιο με βάση το κριτήριο τεμαχισμού `j` περιέχει το σημείο του προγράμματος `i` και `A[i; j] = 0` στην περίπτωση που το τεμάχιο με βάση το κριτήριο τεμαχισμού `j` περιέχει το σημείο του προγράμματος `i`. κατά αυτό τον τρόπο ο πίνακας A καθορίζει το σύνολο των φετών προγράμματος και οι δύο διαστάσεις του πίνακα καθορίζονται από τον αριθμό των σημείων του προγράμματος, για παράδειγμα από τους κόμβους του SDG. Παρακάτω (Εικόνα 5.1) βλέπουμε όλα τα τεμάχια για ένα παράδειγμα αναπαράστασης ενός προγράμματος με n σημεία προγράμματος, σε κάθε περίπτωση παίρνουμε το σημείο προγράμματος σαν κριτήριο τεμαχισμού.

| Program Slicing | Program point | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 4 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 6 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 |
| 7 | 0 | 0 | 1 | 0 | 0 | 1 | 1 | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |

- Συνάρτηση καταλληλότητας: ορίζεται με βάση δύο μετρικές:
    1. Κάλυψη: μετρά το βαθμό που τα σημεία προγράμματος στο σύνολο τεμαχισμού καλύπτουν τα σημεία προγράμματος ολόκληρου του προγράμματος.

2. Επικάλυψη: αποτιμά τον αριθμό των σημείων του προγράμματος που περιέχονται στην τομή ενός σύνολο τεμαχισμού.

Το πλαίσιο εργασίας που παρουσιάστηκε έδωσε ακριβή αποτελέσματα δείχνοντας στην πράξη ότι είναι εφικτή η διατύπωση προβλημάτων ανάλυσης εξαρτήσεων σαν προβλήματα αναζήτησης και ότι δίνονται καλές λύσεις σε λογικό χρόνο με τη χρήση αυτής της τεχνικής.

## Κεφάλαιο 6

# Έλεγχος με χρήση Μετάλλαξης (Mutation Testing)

---



---

## 6.1 Εισαγωγή στον έλεγχο με χρήση μετάλλαξης (Mutation Testing)

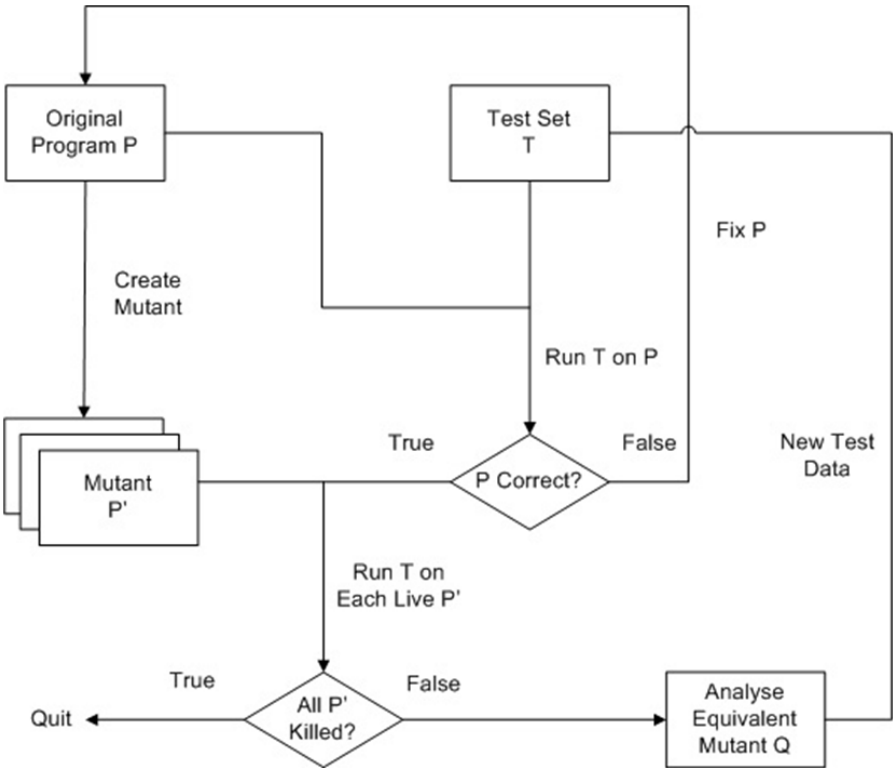
Ο έλεγχος με χρήση μετάλλαξης (mutation testing- M.T.) είναι μία τεχνική ελέγχου την οποία εισήγαγαν οι Hamlet [HA77] και DeMillo [DLS78] στα τέλη της δεκαετίας του 70 και βασίζεται στην εισαγωγή λαθών σε ένα πρόγραμμα. Επιπλέον παρέχει ένα κριτήριο ελέγχου που ονομάζεται «mutation adequacy score». Το εν λόγω κριτήριο χρησιμοποιείται για την μέτρηση της αποτελεσματικότητας ενός συνόλου ελέγχου με βάση την ικανότητα του να εντοπίζει λάθη. Η γενική ιδέα που διέπει τον έλεγχο μετάλλαξης είναι ότι τα λάθη που εισάγονται αντιπροσωπεύουν τα λάθη που γίνονται συχνά από τους προγραμματιστές. Τέτοια λάθη εμφυτεύονται σκόπιμα στο αρχικό πρόγραμμα για τη δημιουργία ενός συνόλου από λανθασμένα προγράμματα τα ονομαζόμενα mutants, κάθε ένα από τα οποία περιέχει μια ξεχωριστή αλλαγή. Για την αποτίμηση της ποιότητας ενός δοθέντος συνόλου ελέγχου τα mutants εκτελούνται με αυτό το σύνολο για να εκλεχθεί κατά πόσο τα εμφυτευμένα λάθη μπορούν να εντοπιστούν.

Ο έλεγχος μετάλλαξης υπόσχεται ότι είναι αποτελεσματικός στην αναγνώριση ικανοποιητικού αριθμού δεδομένων ελέγχου τα οποία μπορούν να χρησιμοποιηθούν για τον εντοπισμό πραγματικών λαθών [GOH92]. Ωστόσο ο αριθμός τέτοιων πιθανών λαθών για ένα δεδομένο πρόγραμμα είναι τεράστιος, είναι λοιπόν αδύνατο να δημιουργηθούν mutants που να τα αντιπροσωπεύουν όλα. Έτσι ο παραδοσιακός

έλεγχος μετάλλαξης στοχεύει μόνο ένα υποσύνολο αυτών των λαθών, αυτά τα λάθη που είναι κοντά στην ορθή έκδοση του προγράμματος. Αυτή η θεωρία βασίζεται σε δύο υποθέσεις: Competent Programmer Hypothesis (CPH) και Coupling Effect [DeMilloLS78].

Η CPH δηλώνει ότι οι προγραμματιστές τείνουν να αναπτύσσουν προγράμματα τα οποία είναι κοντά στη σωστή έκδοση, υποθέτουμε λοιπόν ότι αυτά τα λάθη μπορούν να διορθωθούν με εφαρμογή μερικών συντακτικών αλλαγών. Επομένως στον έλεγχο μετάλλαξης μόνο λάθη που συντελούνται από απλές συντακτικές αλλαγές εφαρμόζονται. Αντίθετα από τη CPH που αφορά την συμπεριφορά του προγραμματιστή το Coupling Effect αφορά τα είδη λαθών που χρησιμοποιούνται στην ανάλυση μετάλλαξης (mutation analysis). Τίθεται λοιπόν η υπόθεση ότι ένα απλό λάθος αναπαριστάται από μια απλή μετάλλαξη, η οποία δημιουργείται με εφαρμογή μιας απλής συντακτικής αλλαγής, ενώ ένα περίπλοκο λάθος αναπαριστάται με μία περίπλοκη μετάλλαξη που δημιουργείται εφαρμόζοντας περισσότερες από μία αλλαγές. Τα περίπλοκα λάθη λοιπόν συσχετίζονται με τα απλά λάθη με τέτοιο τρόπο ώστε ένα σύνολο δεδομένων ελέγχου που εντοπίζει όλα τα απλά λάθη σε ένα πρόγραμμα θα εντοπίσει και ένα μεγάλο ποσοστό των περίπλοκων λαθών. Με αποτέλεσμα οι μεταλλάξεις που χρησιμοποιούνται στον παραδοσιακό έλεγχο με χρήση μεταλλάξεων περιορίζονται σε απλές μόνο μεταλλάξεις.

Στην Εικόνα 6.1 αναπαριστάται ο συνήθης τρόπος ανάλυσης ελέγχου με χρήση μετάλλαξης. Από ένα πρόγραμμα p παράγεται ένα σύνολο από λάθος προγράμματα p', τα προγράμματα αυτά ονομάζονται μεταλλάξεις (mutants) και παράγονται με μερικές απλές αλλαγές στο αρχικό πρόγραμμα p. Ο κανόνας μετασχηματισμού που δημιουργεί μια μετάλλαξη (mutant) από το αρχικό πρόγραμμα λέγεται τελεστής μετάλλαξης (mutation operator). Στη βιβλιογραφία οι τελεστές μετάλλαξης (mutation operators) αναφέρονται επίσης με τις ακόλουθες ονομασίες: «mutant operators», «mutagenic operators», «mutagens and mutation rules» [OU01]. Γενικά οι τελεστές μετάλλαξης αλλάζουν μεταβλητές και εκφράσεις χρησιμοποιώντας αντικατάσταση, εισαγωγή ή διαγραφή τελεστών.




**Εικόνα 6.16 - Συνήθης τρόπος ανάλυσης ελέγχου με χρήση μετάλλαξης [MTR09]**

Στη συνέχεια ένα σύνολο ελέγχου T δίνεται σαν είσοδος στο πρόγραμμα. Πρώτο βήμα πριν ακόμα αρχίσει η ανάλυση μετάλλαξης (mutation analysis) είναι η εκτέλεση του συνόλου ελέγχου με βάση το αρχικό πρόγραμμα p, στην περίπτωση που αυτό δεν εκτελείται με επιτυχία το πρόγραμμα πρέπει να διορθωθεί πριν να τρέξουμε άλλους mutants. Στη συνέχεια κάθε mutant p' θα εκτελεστεί με βάση το σύνολο ελέγχου T. Εάν το αποτέλεσμα της εκτέλεσης του προγράμματος p' είναι διαφορετικό από το αποτέλεσμα της εκτέλεσης του p για ένα σενάριο ελέγχου που ανήκει στο T, τότε το mutant πρόγραμμα p' λέγεται ότι έχει «σκοτωθεί», διαφορετικά λέγεται ότι έχει «επιβιώσει».

Μετά την εκτέλεση όλων των σεναρίων ελέγχου μπορεί να υπάρχουν ακόμα mutants που να έχουν επιβιώσει. Για να καλυτερέψουμε το σύνολο ελέγχου T, το πρόγραμμα ελέγχου μπορεί να παρέχει επιπλέον εισόδους ελέγχου για να «σκοτώσει» τα mutants που έχουν «επιβιώσει». Ωστόσο υπάρχουν mutants που δεν μπορούν να «σκοτωθούν», γιατί πάντα παράγουν το ίδιο αποτέλεσμα με το αρχικό πρόγραμμα. Αυτά τα mutants

ονομάζονται Ισότιμα Mutants (Equivalent Mutants) και παρόλο που είναι συντακτικά διαφορετικά από το αρχικό πρόγραμμα είναι ισοδύναμα λειτουργικά.

Ο έλεγχος με χρήση μετάλλαξης ολοκληρώνεται με τον υπολογισμό του «adequacy score», γνωστού ως Βαθμός Μετάλλαξης (Mutation Score), το οποίο καθορίζει την ποιότητα του συνόλου ελέγχου που δόθηκε σαν είσοδος. Ο βαθμός μετάλλαξης (mutation score-MS) είναι η αναλογία του αριθμού των «σκοτωμένων» mutants προς του συνολικού αριθμού των μη-ισοδύναμων mutants.

Βαθμός Μετάλλαξης (P, T) = KM/(TM-EM)
όπου: KM = αριθμός των «σκοτωμένων» mutants
TM = συνολικός αριθμός των μη-ισοδύναμων mutants
EM = αριθμός των μη-ισοδύναμων mutants

Στόχος της ανάλυσης μετάλλαξης είναι να ανεβάσει το βαθμό μετάλλαξης (mutation score) σε 1, πράγμα που σημαίνει ότι το σύνολο ελέγχου T είναι επαρκές για τον εντοπισμό όλων των λαθών που δηλώνονται από τα mutants.

Παρόλο που ο έλεγχος με χρήση μετάλλαξης μπορεί να αποτιμήσει την ποιότητα ενός συνόλου ελέγχου, το ψηλό υπολογιστικό κόστος εκτέλεσης του μεγάλου αριθμού mutants για ένα σύνολο ελέγχου καθώς και το κόστος ελέγχου του αποτελέσματος του αρχικού προγράμματος με κάθε σενάριο ελέγχου, αποτελούν σοβαρά μειονεκτήματα της μεθόδου. Τέλος χρειάζεται επιπλέον προσπάθεια από τους προγραμματιστές για τον εντοπισμό ισότιμων Mutants (Equivalent Mutants).

Ο αναγνώστης μπορεί να ανατρέξει στον πιο κάτω διαδικτυακό χώρο [MTR09] για περισσότερες πληροφορίες σχετικά με τον έλεγχο με χρήση μετάλλαξης, δημοσιεύσεις που αφορούν το εν λόγω θέμα αλλά και τεχνικές ελέγχου με χρήση μετάλλαξης.

## 6.2 Έλεγχος με χρήση μετάλλαξης (Mutation Testing) για αντικειμενοστρεφή (O-O) προγράμματα σε γλώσσα Java

Τα αντικειμενοστρεφή προγράμματα έχουν αρκετές διαφορές από τα παραδοσιακά προγράμματα, είναι συχνά δομημένα διαφορετικά και περιέχουν χαρακτηριστικά όπως εμφωλιασμός (encapsulation), κληρονομικότητα (inheritance), και πολυμορφισμός (polymorphism). Αυτές οι διαφορές και τα καινούργια χαρακτηριστικά στα αντικειμενοστρεφή προγράμματα αλλάζουν και τις απαιτήσεις του ελέγχου με χρήση μετάλλαξης. Η κύρια διαφορά για τον έλεγχο είναι ότι η αντικειμενοστραφής προσέγγιση αλλάζει τα επίπεδα στα οποία γίνεται ο έλεγχος. Ο έλεγχος προγράμματος και συνένωσης κατηγοριοποιείται σε τέσσερα επίπεδα [HR94, GO04]: 1. μέσα στη μέθοδο (intra-method), 2. μεταξύ μεθόδων (inter-method), 3. μέσα στη κλάση (intra-class) και 4. ανάμεσα σε κλάσεις (inter-class).

- Intra-method Level: σε αυτή την περίπτωση το λάθος συμβαίνει όταν η λειτουργικότητα της μεθόδου υλοποιείται λανθασμένα. Ο έλεγχος μέσα στην κλάση αντιστοιχεί με τον έλεγχο προγράμματος στο συμβατικό προγραμματισμό. Έτσι οι ερευνητές έχουν υποθέσει ότι οι παραδοσιακοί τελεστές μετάλλαξης για τα διαδικασιακά προγράμματα επαρκούν για αυτό το επίπεδο.
- Inter-method Level: σε αυτή την περίπτωση τα λάθη βρίσκονται στη διασύνδεση ανάμεσα σε ζευγάρια μεθόδων της ίδιας κλάσης. Ο έλεγχος σε αυτό το επίπεδο είναι ισότιμος με τον έλεγχο συνένωσης μεθόδων στις διαδικασιακές γλώσσες προγραμματισμού.
- Intra-class Level: σε αυτή την περίπτωση οι έλεγχοι κατασκευάζονται για μια κλάση με σκοπό τον έλεγχο της κλάσης στο σύνολο της. Είναι μια περίπτωση του ελέγχου μονάδας (unit) ή module στον παραδοσιακό προγραμματισμό. Ελέγχει την αλληλεπίδραση των δημόσιων (public) μεθόδων της κλάσης όταν καλούνται σε ξεχωριστή σειρά. Οι έλεγχοι είναι συνήθως αλλεπάλληλες εκτελέσεις των μεθόδων της κλάσης και περικλείουν εκτενείς ελέγχους των δημόσιων διαπροσωπειών (public interfaces) στην κλάση.
- Inter-class Level: σε αυτή την περίπτωση ο έλεγχος εξασκείται σε περισσότερες από μια κλάσεις και επιπλέον εξετάζεται τυχόν ύπαρξη λαθών στην συνένωση

των κλάσεων. Ο έλεγχος είναι ο παραδοσιακός έλεγχος συνένωσης και έλεγχος σε υποσυστήματα που χρησιμοποιούνται σπάνια. Τα περισσότερα λάθη που βρίσκουμε σε αυτή την περίπτωση σχετίζονται με τον πολυμορφισμό , την κληρονομικότητα και την προσπελασιμότητα.

Επιπρόσθετα τίθεται η ανάγκη για τελεστές που να διαχειρίζονται νέους τύπους λαθών που προκύπτουν από τα αντικειμενοστρεφή χαρακτηριστικά. Για τον σκοπό αυτό έχει αναπτυχθεί ένα σύνολο τελεστών class mutation operators που λειτουργεί συμπληρωματικά ως προς τους τελεστές μετάλλαξης σε επίπεδο μεθόδου (method-level) που ισχύουν για τα παραδοσιακά δομημένα προγράμματα [MO05b]. Τέλος ένα σύστημα μετάλλαξης αντικειμενοστρεφούς προγράμματος πρέπει να είναι σε θέση να εξάγει πληροφορίες και να εκτελεί προγράμματα με βάση την αντικειμενοστρεφή σκοπιά. Για παράδειγμα πρέπει να διαχειρίζονται τύποι που ορίζονται από τον προγραμματιστή, για παράδειγμα κλάσεις και αναφορές σε τέτοιους τύπους. Πρέπει επίσης να ληφθούν υπόψη ο οι σχέσεις ελέγχου, δεδομένων, κληρονομικότητας και πολυμορφισμού μεταξύ των συστατικών. Παρακάτω θα δούμε το σύνολο των τελεστών μετάλλαξης για Java προγράμματα όπως έχει καθοριστεί στο [MKO02].

| Language Feature | Operator | Description |
|---|---|---|
| Access Control | AMC | Access modifier change |
| Inheritance | IHD | Hiding variable deletion |
| | IHI | Hiding variable insertion |
| | IOD | Overriding method deletion |
| | IOP | overriding method calling position change |
| | IOR | Overriding method rename |
| | ISK | *super* keyword deletion |
| | IPC | Explicit call of a parent's constructor deletion |
| Polymorphism | PNC | *new* method call with child class type |
| | PMD | Instance variable declaration with parent class type |
| | PPD | Parameter variable declaration with child class type |
| | PRV | Reference assignment with other comparable type |
| Overloading | OMR | Overloading method contents change |
| | OMD | Overloading method deletion |
| | OAO | Argument order change |
| | OAN | Argument number change |
| Java-Specific Features | JTD | *this* keyword deletion |
| | JSC | *static* modifier change |
| | JID | Member variable initialization deletion |
| | JDC | Java-supported default constructor creation |
| Common Programming Mistakes | EOA | Reference assignment and content assignment replacement |
| | EOC | Reference comparison and content comparison replacement |
| | EAM | Accessor method change |
| | EMM | Modifier method change |

**Εικόνα 6.17 - Τελεστές μετάλλαξης για Inter-Class έλεγχο [MKO02]**

**1. Απόκρυψη πληροφορίας (Information Hiding (Access Control))**: ο έλεγχος πρόσβασης είναι μια συχνή πηγή λαθών σε αντικειμενοστραφή προγράμματα. Η σημασιολογία των διαφόρων επιπέδων προσβασιμότητας δεν γίνεται συχνά κατανοητή και η προσβασιμότητα σε μεταβλητές και μεθόδους δεν λαμβάνεται πάντα υπόψη κατά τον σχεδιασμό. Ο φτωχός ορισμός των επιπέδων προσβασιμότητας δεν δημιουργεί κατ' ανάγκη προβλήματα, αλλά κατά τη συνένωση της κλάσης με άλλες κλάσεις ή την τροποποίηση της μπορούν να προκύψουν προβλήματα. Ο τελεστής μετάλλαξης AMC έχει οριστεί για την εν λόγω κατηγορία.

**2. Κληρονομικότητα (Inheritance)**: αν και η κληρονομικότητα είναι ένας αποτελεσματικός και χρήσιμος μηχανισμός αφαιρετικότητας, αν χρησιμοποιηθεί λανθασμένα μπορούν να προκύψουν πολλαπλά προβλήματα. Έχουν οριστεί επτά τελεστές μετάλλαξης για τον έλεγχο των διαφορετικών σκοπιών της χρήσης της κληρονομικότητας, που αφορούν variable hiding, method overriding, τη χρήση του super, και ορισμό constructors.

**3. Πολυμορφισμός (Polymorphism)**: ο πολυμορφισμός και το dynamic binding επιτρέπουν στις αναφορές αντικειμένων να αλλάζουν τύπο σε διαφορετικές εκτελέσεις ή σε διαφορετικές στιγμές κατά την ίδια εκτέλεση. Έτσι αναφορές αντικειμένων μπορεί να αναφέρονται σε αντικείμενα των οποίων ο πραγματικός τύπος διαφέρει από τον τύπο που έχουν οριστεί. Σε πολλές γλώσσες συμπεριλαμβανομένης και της Java, ο πραγματικός τύπος μπορεί να είναι οποιοσδήποτε τύπος ο οποίος αποτελεί υποκλάση του τύπου στον οποίο έχουν οριστεί. Ο πολυμορφισμός επιτρέπει σε ένα αντικείμενο να έχει διαφορετική συμπεριφορά ανάλογα του actual type. Έχουν οριστεί τέσσερις τελεστές για αυτή τη κατηγορία.

**4. Υπερφόρτωση (Overloading)**: η υπερφόρτωση μεθόδου επιτρέπει σε δύο ή περισσότερες μεθόδους της ίδιας κλάσης ή τύπου να μοιράζονται το ίδιο όνομα φτάνει να έχουν διαφορετικούς τύπους ορισμάτων. Όπως και με τον πολυμορφισμό, είναι σημαντικό κατά τον έλεγχο να διασφαλιστεί ότι το κάλεσμα μιας μεθόδου καλεί την σωστή μέθοδο με τις κατάλληλες παραμέτρους. Για αυτή την ομάδα έχουν οριστεί τέσσερις τελεστές μετάλλαξης.

**5. Χαρακτηριστικά της γλώσσας Java (Java-Specific Features)**: μιας και ο έλεγχος με χρήση μετάλλαξης εξαρτάται από τη γλώσσα προγραμματισμού οι τελεστές μετάλλαξης πρέπει να συμφωνούν με συγκεκριμένα χαρακτηριστικά της γλώσσας προγραμματισμού. Η Java έχει μερικά χαρακτηριστικά τα οποία δεν υπάρχουν σε άλλες αντικειμενοστρεφείς γλώσσες για το σκοπό αυτό έχουν οριστεί τέσσερις τελεστές για την διασφάλιση της σωστής χρήσης αυτών των χαρακτηριστικών.

**6. Κοινά προγραμματιστικά λάθη (Common Programming Mistakes)**: αυτή η κατηγορία έχει σκοπό να εντοπίσει τυπικά λάθη που γίνονται από τους προγραμματιστές αντικειμενοστραφούς λογισμικού.
Τέσσερις τελεστές έχουν οριστεί σε αυτή την κατηγορία.

## 6.3 Εργαλείο muJava και τελεστές μετάλλαξης (mutation operators) που χρησιμοποιεί

Έχουν προταθεί διάφορα εργαλεία για διεξαγωγή ελέγχου με χρήση μετάλλαξης, στη συνέχεια θα δούμε ενδεικτικά μερικά από αυτά. Ένα από αυτά είναι και το Mothra [DS86], που είναι ένα ευέλικτο και πλήρες περιβάλλον ελέγχου το οποίο έχει αναπτυχθεί για τη Fortran 77. Αν και δεν μπορεί να χρησιμοποιηθεί με Java λογισμικό, είναι το πρώτο ολοκληρωμένο σύστημα ελέγχου με χρήση μετάλλαξης.

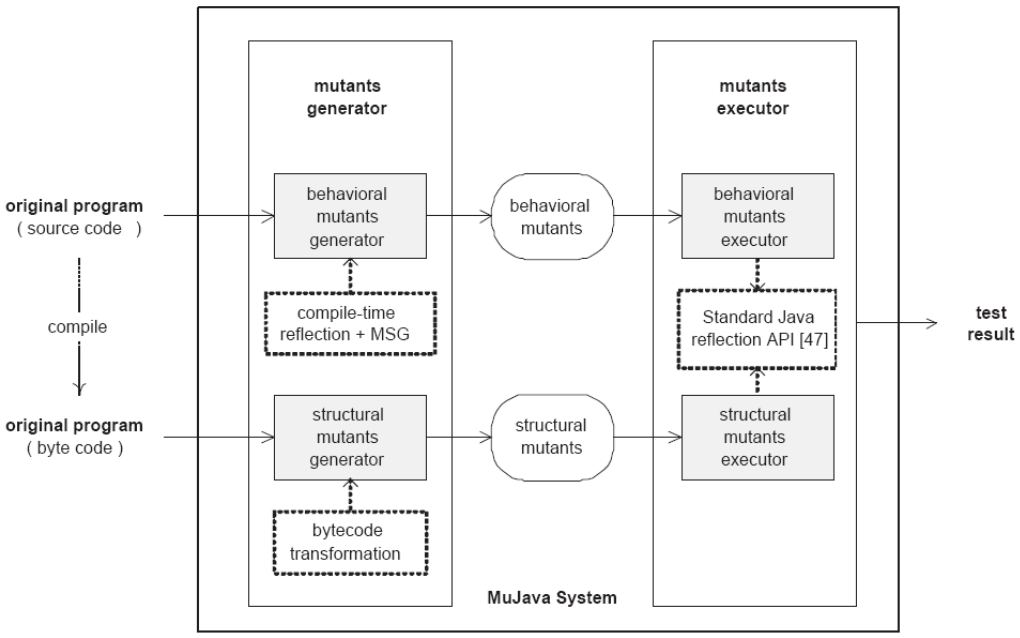
Το Jester [MO00] είναι ένα απλό open-source εργαλείο ελέγχου με χρήση μετάλλαξης για Java. Σχεδιάστηκε για να εκτελεί απλούς τελεστές μετάλλαξης σε Java προγράμματα και να λειτουργεί συμπληρωματικά με το γνωστό πλαίσιο εργασίας JUnit το οποίο εξασκεί έλεγχο σε κομμάτια κώδικα εξάγοντας σενάρια ελέγχου. Η μετάλλαξη του κώδικα απαιτεί επαναγλώτιση των αντίστοιχων κλάσεων για κάθε σημείο μετάλλαξης, το οποίο είναι χρονοβόρο. Επιπλέον η προσθήκη μεταλλάξεων απευθείας στον κώδικα μπορεί να είναι εσφαλμένη μιας και έχουν καταγραφεί περιπτώσεις στις οποίες ο Jester δημιουργεί μεταλλάξεις σε σχόλια του κώδικα και όχι στον κώδικα καθ' αυτό. Καθώς το Jester δεν εφαρμόζει κάποιο εξεζητημένο αλγόριθμο για τον έλεγχο με χρήση μετάλλαξης ώστε να επιταχύνει την διαδικασία, είναι πολύ αργός. Επιπρόσθετα το εύρος των μεταλλάξεων που εφαρμόζει είναι περιορισμένο και δεν υπερτερεί σε μεγάλο βαθμό από τα εργαλεία που αξιολογούν τα σενάρια ελέγχου ως προς το

ποσοστό κάλυψης του κώδικα (code coverage tools). Καταλήγοντας έχει αποδειχτεί ότι το Jester ήταν περιορισμένο, αναποτελεσματικό και μη-πρακτικό για χρήση σε μεγάλα προγράμματα.

Τέλος το MuJava [OMK05] είναι ένα σύστημα ελέγχου με χρήση μετάλλαξης για έλεγχο Java προγραμμάτων. Πρωταρχικός του σκοπός ήταν να μελετήσει τελεστές σχετικούς με γλώσσες αντικειμενοστρεφούς προγραμματισμού. Η χρήση του σε μεγάλα προγράμματα αποτελεί πρόκληση μιας και οι τελεστές μετάλλαξης που χρησιμοποιεί βρίσκονται στο ανώτατο επίπεδο τεχνικής, μέχρι στιγμής αποτελεί το εργαλείο για έλεγχο με χρήση μετάλλαξης που χρησιμοποιεί το πιο ολοκληρωμένο σύνολο τελεστών.

Το MuJava παράγει μεταλλάξεις τόσο για τον παραδοσιακό έλεγχο όσο και για έλεγχο στο επίπεδο κλάσεων, συνδυάζοντας δύο βασικές τεχνολογίες Mutant Schemata Generation (MSG) [UOH93] και bytecode μετάφραση. Χρησιμοποιεί λοιπόν την υπάρχουσα μέθοδο MSG για την παραγωγή «meta-mutant» προγράμματος στο επίπεδο source, το οποίο ενσωματώνει πολλές μεταλλάξεις. Το εργαλείο MuJava εργάζεται απευθείας στο bytecode, κατά αυτό τον τρόπο χρειάζονται μόνο δύο μεταγλωττίσεις, μεταγλώττιση του αρχικού προγράμματος και μεταγλώττιση των metamutants που δημιουργούνται με την MSG. Αυτός ο σχεδιασμός λειτουργεί υπέρ της καλυτέρεψης της επίδοσης σε σχέση με συστήματα μετάλλαξης τα οποία μεταγλωττίζουν όλα τα mutants ή με παλαιότερα εργαλεία σαν το Mothra [DS86] το οποίο αλλάζει ένα ενδιάμεσο κώδικα που σχεδιάζεται ειδικά για τις μεταλλάξεις. Η μετάφραση σε MSG και bytecode είναι σημαντικά πιο γρήγορη.

Στην Εικόνα 6.3 βλέπουμε τη γενική δομή του εργαλείου. Το MuJava υλοποιεί τόσο τους τελεστές μετάλλαξης inter-class όσο και intra-class.




**Εικόνα 6.18 – MuJava [OMK05]**

### Τελεστές μετάλλαξης σε επίπεδο μεθόδου (method-level)

Στη βιβλιογραφία έχουν βρεθεί δύο εργασίες που ασχολούνται με τους τελεστές μετάλλαξης του muJava εργαλείου [MO05, MO05b]. Στην [MO05b] παρουσιάζονται οι τελεστές μετάλλαξης σε επίπεδο μεθόδου (method-level) για Java που χρησιμοποιούνται στο muJava. Κατά τον σχεδιασμό των method-level τελεστών μετάλλαξης για Java, ακολουθείται η επιλεκτική προσέγγιση [OLRUZ96]. Για τις μεταλλάξεις του muJava χρησιμοποιούνται τελεστές που αλλάζουν τις δηλώσεις με αντικαταστάσεις (replacing), αφαιρέσεις (deleting), και προσθέσεις (inserting) στοιχειωδών τελεστών (primitive operators). Το muJava παρέχει έξι τύπους στοιχειωδών τελεστών (primitive operators): (1) αριθμητικό τελεστή (arithmetic operator), (2) σχεσιακό τελεστή (relational operator), (3) τελεστή συνθήκης (conditional operator), (4) shift operator, (5) λογικό τελεστή (logical operator) και (6) ανάθεση (assignment). Στον πίνακα της Εικόνα 6.4 παραθέτονται δώδεκα τελεστές σε επίπεδο μεθόδου (method-level). Επιπρόσθετα, μερικοί από τους τελεστές υποδιαιρούνται σε δύο ή τρεις άλλους τελεστές ανάλογα με τον αριθμό και τον τύπο των τελεστών. Για παράδειγμα ο τελεστής AOR υποδιαιρείται στους: AORB (binary), AORU (unary), και AORS (short-cut).

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

**Εικόνα 6.19 - Τελεστές μετάλλαξης Method-level για Java [MO05b]**

### 1. Αριθμητικοί Τελεστές (Arithmetic Operators)

Η γλώσσα προγραμματισμού Java υποστηρίζει πέντε δυαδικούς αριθμητικούς τελεστές για όλους τους ακέραιους και δεκαδικούς αριθμούς: (1) `+`, (2) `-`, (3) `*`, (4) `/` και (5) `%`. Επιπλέον υποστηρίζει τέσσερις τύπους short-cut αριθμητικών τελεστών: (1) `op++`, (2) `++op`, (3) `op--` και (4) `--op`.

- $AOR_B$: Αντικατάσταση Αριθμητικού Τελεστή (Arithmetic Operator Replacement).
  Αντικατάστησε βασικούς δυαδικούς αριθμητικούς τελεστές με άλλους δυαδικούς αριθμητικούς τελεστές.
- $AOR_U$: Αντικατάσταση Αριθμητικού Τελεστή (Arithmetic Operator Replacement)
  Αντικατάστησε βασικούς μοναδιαίους (unary) αριθμητικούς τελεστές με άλλους μοναδιαίους αριθμητικούς τελεστές.
- $AOR_S$: Αντικατάσταση Αριθμητικού Τελεστή (Arithmetic Operator Replacement)
  Αντικατάστησε short-cut αριθμητικούς τελεστές με άλλους μοναδιαίους (unary) αριθμητικούς τελεστές.
- $AOI_U$ : Προσθήκη Αριθμητικού Τελεστή (Arithmetic Operator Insertion)
  Πρόσθεσε βασικούς μοναδιαίους (unary) αριθμητικούς τελεστές.

- AOI$_S$: Προσθήκη Αριθμητικού Τελεστή (Arithmetic Operator Insertion)
  Πρόσθεσε short-cut αριθμητικούς τελεστές.
- AOD$_U$: Διαγραφή Αριθμητικού Τελεστή (Arithmetic Operator Deletion)
  Διέγραψε βασικούς μοναδιαίους (unary) αριθμητικούς τελεστές.
- AOD$_S$: Διαγραφή Αριθμητικού Τελεστή (Arithmetic Operator Deletion)
  Διέγραψε short-cut αριθμητικούς τελεστές.

**2. Σχεσιακοί Τελεστές (Relational Operators)**

Η Java παρέχει έξι είδη σχεσιακών τελεστών: (1) >, (2) >=, (3) <, (4) <=, (5) == και (6) ! =. Λόγω του ότι αυτοί οι τελεστές δέχονται δύο τελεστέους (operands), μόνο η αντικατάσταση επιτρέπεται στην περίπτωση των σχεσιακών τελεστών.

- ROR: Αντικατάσταση Σχεσιακού Τελεστή (Relational Operator Replacement)
  Αντικατέστησε σχεσιακούς τελεστές με άλλους σχεσιακούς τελεστές.

**3. Υποθετικοί Τελεστές (Conditional Operators)**

Η γλώσσα προγραμματισμού Java υποστηρίζει έξι υποθετικούς τελεστές (conditional operators) από τους οποίους οι πέντε είναι δυαδικοί (binary) και ο ένας μοναδιαίος (unary). Οι πέντε δυαδικοί υποθετικοί τελεστές είναι: (1) &&, (2) ||, (3) &, (4) | και (5) ^. Ο μοναδιαίος υποθετικός τελεστής είναι: '!'.

- COR: Αντικατάσταση Υποθετικού Τελεστή (Conditional Operator Replacement)
  Αντικατέστησε δυαδικούς υποθετικούς τελεστές με άλλους δυαδικούς υποθετικούς τελεστές.
- COI: Προσθήκη Υποθετικού Τελεστή (Conditional Operator Insertion)
  Πρόσθεσε ένα μοναδιαίο (unary) τελεστή συνθήκης .
- COD: Διαγραφή Υποθετικού Τελεστή (Conditional Operator Deletion)
  Διέγραψε τον μοναδιαίο (unary) τελεστή συνθήκης .

4. **Τελεστής Shift (Shift Operators)**

Η Java παρέχει τρεις τελεστές shift: (1) >>, (2) <<, και (3) >>>>. Ένας τελεστής shift εκτελεί επεξεργασία ψηφίων σε δεδομένα αλλάζοντας τα ψηφία του πρώτου τελεστή προς τα αριστερά ή δεξιά. Μόνο η αντικατάσταση επιτρέπεται για τους τελεστές shift.

- SOR: η αντικατάσταση του τελεστή Shift ανταλλάζει τους τελεστές shift με άλλους τελεστές shift.

5. **Λογικοί Τελεστές (Logical Operators)**

Η Java παρέχει τέσσερις λογικούς τελεστές για την εκτέλεση bitwise λειτουργιών στους τελεστές τους. Οι τρεις από αυτούς είναι δυαδικοί: (1) &, (2) |, and (3) ^ και ο ένας είναι μοναδιαίος: ~.

- LOR: Αντικατάσταση Λογικού Τελεστή (Logical Operator Replacement)
  Αντικαθιστά δυαδικούς λογικούς τελεστές με άλλους δυαδικούς τελεστές.
- LOI: Προσθήκη Λογικού Τελεστή (Logical Operator Insertion)
  Πρόσθεσε ένα μοναδιαίο λογικό τελεστή.
- LOD: Διαγραφή Λογικού Τελεστή (Logical Operator Delete)
  Διέγραψε ένα μοναδιαίο λογικό τελεστή.

6. **Τελεστές Ανάθεσης (Assignment Operators)**

Ο βασικός τελεστής ανάθεσης αναθέτει τη τιμή του δεξιού μέλους της έκφρασης (op2) στη μεταβλητή που βρίσκεται στο αριστερό μέλος (op1). Συμπληρωματικά με τη βασική ανάθεση η γλώσσα προγραμματισμού Java ορίζει έντεκα short-cut τελεστές ανάθεσης οι οποίοι εκτελούν μια πράξη και μια ανάθεση με χρήση ενός τελεστή:
(1) +=, (2) -=, (3) *=, (4) /=, (5) %=, (6) &=, (7) |=, (8) ^=, (9) <<=, (10) >>= και (11) >>>=.

- ASRS : Αντικατάσταση Short-Cut Τελεστή Ανάθεσης (Short-Cut Assignment Operator Replacement)

Αντικαθιστά τους short-cut τελεστές ανάθεσης με άλλους short-cut τελεστές του ίδιου τύπου.

**Τελεστές μετάλλαξης κλάσης (class mutation operators)**

Οι τελεστές μετάλλαξης κλάσης (class mutation operators) που χρησιμοποιούνται από το muJava περιγράφονται στην [MO05] και κατηγοριοποιούνται στις ακόλουθες τέσσερις ομάδες με βάση τα χαρακτηριστικά της γλώσσας που επηρεάζουν, οι κατηγοριοποίηση παρουσιάζεται στον πίνακα της Εικόνα 6.5. Οι τρεις πρώτες κατηγορίες βασίζονται σε εκείνα τα χαρακτηριστικά που είναι κοινά για όλες τις αντικειμενοστρεφής γλώσσες, ενώ η τελευταία κατηγορία περιλαμβάνει τα αντικειμενοστρεφή χαρακτηριστικά που σχετίζονται με τη Java (Java-specific).

1. Encapsulation
2. Κληρονομικότητα (Inheritance)
3. Πολυμορφισμός (Polymorphism)
4. Java-Specific Features

| Language Feature | Operator | Description |
|---|---|---|
| Encapsulation | AMC | Access modifier change |
| Inheritance | IHD | Hiding variable deletion |
| | IHI | Hiding variable insertion |
| | IOD | Overriding method deletion |
| | IOP | Overriding method calling position change |
| | IOR | Overriding method rename |
| | ISI | super keyword insertion |
| | ISD | super keyword deletion |
| | IPC | Explicit call to a parent's constructor deletion |
| Polymorphism | PNC | new method call with child class type |
| | PMD | Member variable declaration with parent class type |
| | PPD | Parameter variable declaration with child class type |
| | PCI | Type cast operator insertion |
| | PCC | Cast type change |
| | PCD | Type cast operator deletion |
| | PRV | Reference assignment with other comparable variable |
| | OMR | Overloading method contents replace |
| | OMD | Overloading method deletion |
| | OAC | Arguments of overloading method call change |
| Java-Specific Features | JTI | this keyword insertion |
| | JTD | this keyword deletion |
| | JSI | static modifier insertion |
| | JSD | static modifier deletion |
| | JID | Member variable initialization deletion |
| | JDC | Java-supported default constructor creation |
| | EOA | Reference assignment and content assignment replacement |
| | EOC | Reference comparison and content comparison replacement |
| | EAM | Acessor method change |
| | EMM | Modifier method change |

**Εικόνα 6.20 - Τελεστές μετάλλαξης για Inter-Class έλεγχο [MO05]**

## 1. Encapsulation

- AMC - Access modifier change: Ο τελεστής AMC αλλάζει το επίπεδο ελέγχου για τις instance μεταβλητές και μεθόδους σε άλλα επίπεδα ελέγχου. Σκοπός του τελεστή AMC είναι να καθοδηγήσει τους προγραμματιστές στη δημιουργία σεναρίων ελέγχου τα οποία εξασφαλίζουν ότι η προσβασιμότητα (accessibility) είναι ορθή.

| **Αρχικό Πρόγραμμα** | **AMC Mutants** |
|---|---|
| public Stack s; | private Stack s;<br>protected Stack s;<br>Stack s; |

## 2. Inheritance

- IHD – Διαγραφή κρυμμένης μεταβλητής (Hiding variable deletion): Ο IHD τελεστής διαγράφει μια κρυμμένη μεταβλητή (hiding variable), δηλαδή μια μεταβλητή σε μια υποκλάση η οποία έχει το ίδιο όνομα και τύπο με μια μεταβλητή στην κλάση πατέρα. Αυτό έχει σαν αποτέλεσμα οι αναφορές σε αυτή τη μεταβλητή να έχουν πρόσβαση στη μεταβλητή που έχει οριστεί στον πατέρα.

| **Αρχικό Πρόγραμμα** | **IHD Mutants** |
|---|---|
| class List { | class List { |
| int size; | int size; |
| ... ... | ... ... |
| } | } |
| class Stack extends List { | class Stack extends List { |
| int size; | // int size; |
| ... ... | ... ... |
| } | } |

- IHI – Προσθήκη κρυμμένης μεταβλητής (Hiding variable insertion): Ο IHI τελεστής προσθέτει μια κρυμμένη μεταβλητή σε μια υποκλάση.

| **Αρχικό Πρόγραμμα** | **IHI Mutants** |
|---|---|
| class List { | class List { |
| int size; | int size; |
| ... ... | ... ... |
| } | } |
| class Stack extends List { | class Stack extends List { |
| | int size; |
| ... ... | ... ... |
| } | } |

- IOD – Διαγραφή μεθόδου Overriding (Overriding method deletion): Ο τελεστής IOD διαγράφει ολόκληρη τη δήλωση μιας overriding μεθόδου σε μια υποκλάση έτσι που αναφορές στη μέθοδο να χρησιμοποιούν τη μέθοδο του πατέρα.

| **Αρχικό Πρόγραμμα** | **IOD Mutants** |
|---|---|
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| void push (int a) { ... } | // void push (int a) { ... } |
| } | } |

- IOP – Μετακίνηση της κλήσης της μεθόδου Overridden (Overridden method calling position change): Σε μερικές περιπτώσεις μια overriding μέθοδος στη κλάση παιδί χρειάζεται να καλέσει τη μέθοδο που υπερκαλύπτει (overrides) στην κλάση πατέρας. Αυτό συμβαίνει εάν η μέθοδος του πατέρα χρησιμοποιεί μια ιδιωτική (private) μεταβλητή v, πράγμα που σημαίνει ότι η μέθοδος στη κλάση παιδί μπορεί να μην αλλάζει την v απευθείας. Αν η κλήση στην εκδοχή του πατέρα δεν γίνει την κατάλληλη στιγμή μπορεί να προκληθεί λανθασμένη συμπεριφορά. Ο IOP τελεστής μετακινεί την κλήση στη overridden μέθοδο στην πρώτη και τελευταία δήλωση της μεθόδου και μια δήλωση πάνω ή κάτω.

| **Αρχικό Πρόγραμμα** | **IOP Mutants** |
|---|---|
| class List { } | class List { |
| ... ... | ... ... |
| void SetEnv() | void SetEnv() |
| {size = 5; ... } | {size = 5; ... } |
| } | } |
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| void SetEnv() { | void SetEnv() { |
| super.SetEnv(); | size = 10; |
| size = 10; | super.SetEnv(); |
| } | } |
| } | } |

- IOR – Αλλαγή ονόματος της Overridden μεθόδου (Overridden method rename): Ο τελεστής IOR αλλάζει το όνομα της μεθόδου στον πατέρα έτσι που η υπερκάλυψη (overriding) να μην επηρεάζει την μέθοδο στον πατέρα.

| **Αρχικό Πρόγραμμα** | **IOR Mutants** |
|---|---|
| class List { } | class List { |
| ... ... | ... ... |
| void f() { … } | void f'() { … } |
| void m() {... f(); ... } | void m() {... f'(); ... } |
| } | } |
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| void f() { ... } | void f() { ... } |
| void g() {... f(); ... } | void g() {... f(); ... } |
| } | } |

Η λέξη κλειδί `super` χρησιμοποιείται όταν θέλουμε να αναφερθούμε στις μεταβλητές ή μεθόδους του πατέρα μέσα από την κλάση παιδί. Η χρήση της λέξης κλειδί `super` πρέπει να γίνεται με προσοχή.

- ISI – Προσθήκη της λέξης κλειδί `super` (super keyword insertion): Ο ISI τελεστής προσθέτει τη λέξη κλειδί `super` έτσι που μια αναφορά στη τιμή ή μεταβλητή σημαίνει το overridden στιγμιότυπο της μεταβλητής ή της μεθόδου.

| **Αρχικό Πρόγραμμα** | **ISI Mutants** |
|---|---|
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| int MyPop( ) { | int MyPop( ){ |
| ... ... | ... ... |
| return val*num; | return val*super.num; |
| } | } |
| } | } |

- ISD - Διαγραφή της λέξης κλειδί `super` (`super` keyword deletion): Ο τελεστής ISD διαγράφει τη λέξη κλειδί `super`.

| **Αρχικό Πρόγραμμα** | **ISD Mutants** |
|---|---|
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| int MyPop( ) { | int MyPop( ){ |
| ... ... | ... ... |
| return val*super.num; | return val*num; |
| } | } |
| } | } |

Η κλάση παιδί μπορεί να χρησιμοποιήσει τη λέξη κλειδί `super` για να καλέσει ένα συγκεκριμένο constructor της κλάσης πατέρα.

- IPC – Διαγραφή του καλέσματος του constructor του πατέρα: ο τελεστής IPC διαγράφει `super` constructor calls, έτσι ώστε να καλεστεί ο default constructor της κλάσης παιδί.

| **Αρχικό Πρόγραμμα** | **IPC Mutants** |
|---|---|
| class Stack extends List { | class Stack extends List { |
| ... ... | ... ... |
| Stack (int a) { | Stack (int a) { |
| super (a); | // super (a); |
| } | } |
| } | } |

3. Πολυμορφισμός (Polymorphism)

- PNC – κλήση της μεθόδου new για τον τύπο του παιδιού.

| **Αρχικό Πρόγραμμα** | **PNC Mutants** |
|---|---|
| Parent a; | Parent a; |

| | |
|---|---|
| a = new Parent(); | a = new Child; |

- PMD – Ορισμός μεταβλητής με τον τύπο της κλάσης πατέρας.

| **Αρχικό Πρόγραμμα** | **PMD Mutants** |
|---|---|
| Child b; | Parent b; |
| b = new Child(); | a = new Child; |

- PPD – Ορισμός μεταβλητής παραμέτρου με βάση τον τύπο της κλάσης παιδί.

| **Αρχικό Πρόγραμμα** | **PPD Mutants** |
|---|---|
| boolean equals (Child o) { ... } | boolean equals (Parent o) { ... } |

- PCI – Προσθήκη Type cast τελεστή: Ο τελεστής PCI αλλάζει τον actual τύπο της αναφοράς αντικειμένου (object reference) σε αυτό του πατέρα ή του παιδιού του πρωτότυπου τύπου ορισμού.

| **Αρχικό Πρόγραμμα** | **PCI Mutants** |
|---|---|
| Child cRef; | Child cRef; |
| Parent pRef = cRef; | Parent pRef = cRef; |
| pRef.toString(); | ((Child)pRef).toString(); |

- PCD – Διαγραφή Type cast τελεστή.

| **Αρχικό Πρόγραμμα** | **PCD Mutants** |
|---|---|
| Child cRef; | Child cRef; |
| Parent pRef = cRef; | Parent pRef = cRef; |
| ((Child)pRef).toString(); | pRef.toString(); |

- PCC – Αλλαγή Cast type: Ο τελεστής PCC αλλάζει τον τύπο στον οποίο θα γίνει cast μια μεταβλητή.

| Αρχικό Πρόγραμμα | PCC Mutants |
|---|---|
| ((Parent)ref).toString(); | ((Child)ref).toString(); |

- PRV – Αλλαγή assignment με άλλο συμβατό τύπο: Στο παράδειγμα η μεταβλητή `obj` είναι τύπου Object, και στον αρχικό κώδικα της ανατίθεται ένα αντικείμενο τύπου String. Στον κώδικα μετάλλαξης, ανατίθεται στην μεταβλητή ένα αντικείμενο τύπου Integer.

| Αρχικό Πρόγραμμα | PRV Mutants |
|---|---|
| Object obj; | Object obj; |
| String s = \Hello"; | String s = \Hello"; |
| Integer i = new Integer(4); | Integer i = new Integer(4); |
| obj = s; | obj = i; |

- OAC – Αλλαγή ορίσματος μεθόδου υπερκάλυψης (overloading method): Ο τελεστής OAC αλλάζει την σειρά ή τον αριθμό των ορισμάτων κατά την κλήση μιας μεθόδου στις περιπτώσεις που υπάρχει μια overloading μέθοδος που μπορεί να δεχτεί την καινούργια λίστα ορισμάτων, με αυτό τον τρόπο θα καλεστεί διαφορετική μέθοδος.

| Αρχικό Πρόγραμμα | OAC Mutants |
|---|---|
| s.Push (0.5, 2); | s.Push (2, 0.5); |
| | s.Push (2); |
| | s.Push (0.5); |
| | s.Push (); |

## 4. Java-Specific Features

- JTI – Προσθήκη της δεσμευμένης λέξης `this`.

| Αρχικό Πρόγραμμα | JTI Mutants |
|---|---|

```
class Stack {                          class Stack {
    int size;                              int size;
    ... ...                                ... ...
    void setSize (int size) {              void setSize (int size) {
        this.size=size;                        this.size=this.size;
    }                                      }
}                                      }
```

- JTD – Διαγραφή της δεσμευμένης λέξης `this`.

| **Αρχικό Πρόγραμμα** | **JTD Mutants** |
|---|---|
| class Stack {<br>  int size;<br>  ... ...<br>  void setSize (int size) {<br>  this.size=size;<br>  }<br>} | class Stack {<br>  int size;<br>  ... ...<br>  void setSize (int size) {<br>  size=size;<br>  }<br>} |

- JSI – Προσθήκη του τροποποιητή `static`.

| **Αρχικό Πρόγραμμα** | **JSI Mutants** |
|---|---|
| public int s = 100; | public static int s = 100; |

- JSD – του τροποποιητή `static`.

| **Αρχικό Πρόγραμμα** | **JSD Mutants** |
|---|---|
| public static int s = 100; | public int s = 100; |

- JID – Διαγραφή της αρχικοποίησης μιας member μεταβλητής.

| **Αρχικό Πρόγραμμα** | **JID Mutants** |
|---|---|

| | |
|---|---|
| class Stack { | class Stack { |
| int size = 100; | int size; |
| ... ... | ... ... |
| Stack() { ... ... } | Stack() { ... ... } |
| } | } |

- EOA – Αντικατάσταση reference assignment και content assignment: Ένα κοινό προγραμματιστικό λάθος στη Java είναι η χρήση αναφοράς αντικειμένου (object reference) αντί του περιεχομένου του αντικειμένου που αναφέρει ο δείκτης. Ο τελεστής EOA αντικαθιστά μια ανάθεση ενός pointer reference με ένα copy του αντικειμένου χρησιμοποιώντας τη μέθοδο `clone()`. Η μέθοδος `clone()` αντιγράφει τα περιεχόμενα ενός αντικειμένου, δημιουργώντας και επιστρέφοντας μια αναφορά σε ένα νέο αντικείμενο.

| **Αρχικό Πρόγραμμα** | **EOA Mutants** |
|---|---|
| s1 = new Stack(); | s1 = new Stack(); |
| s2 = s1; | s2 = s1.clone(); |

- EOC – Αντικατάσταση reference comparison με content comparison.

| **Αρχικό Πρόγραμμα** | **EOC Mutants** |
|---|---|
| Integer i1 = new Integer (7); | Integer i1 = new Integer (7); |
| Integer i2 = new Integer (7); | Integer i2 = new Integer (7); |
| boolean b = (i1==i2); | boolean b = (i1.equals (i2)); |

- EAM – Αλλαγή της μεθόδου επανάκτησης (accessor): ο τελεστής EAM αλλάζει το όνομα μιας μεθόδου επανάκτησης (accessor method) αντικαθιστώντας το με το όνομα μιας άλλης accessor μεθόδου η οποία έχει το ίδιο signature.

| **Αρχικό Πρόγραμμα** | **EAM Mutants** |
|---|---|
| point.getX(); | point.getY(); |

- EMM – Αλλαγή μεθόδου τροποποίησης (modifier method): Ο τελεστής EMM εφαρμόζει την ίδια λογική με τον τελεστή EMM, με τη διαφορά ότι δουλεύει με modifier μεθόδους αντί με accessor μεθόδους.

| **Αρχικό Πρόγραμμα** | **EAM Mutants** |
|---|---|
| point.setX (2); | point.setY (2); |

Στο επόμενο κεφάλαιο θα δούμε με λεπτομέρεια πως το εργαλείο muJava και οι τελεστές του χρησιμοποιήθηκαν για τους σκοπούς της παρούσας εργασίας.

## Κεφάλαιο 7

## Εντοπισμός Σφάλματος με χρήση δυναμικού τεμαχισμού προγράμματος, mutation testing και γενετικών αλγορίθμων

---



---

## 7.1 Περιγραφή γενικής ιδέας

**Περιγραφή προβλήματος**

Έχοντας δεδομένο ένα πρόγραμμα το οποίο περιέχει ένα λάθος θέλουμε να εντοπίσουμε τη συγκεκριμένη γραμμή ή ένα πολύ μικρό αριθμό γραμμών του προγράμματος που περιέχουν το λάθος. **Επιπλέον θέλουμε να εντοπίσουμε το λάθος αυτής της γραμμής και να προτείνουμε την διόρθωση του** Θεωρούμε ότι ένα πρόγραμμα περιέχει λάθος αν σε τουλάχιστον μια εκτέλεση του προγράμματος με συγκεκριμένες τιμές εισόδου η τιμή μιας συγκεκριμένης μεταβλητής κατά τον τερματισμό του προγράμματος ή σε οποιαδήποτε προηγούμενη γραμμή του προγράμματος έχει διαφορετική τιμή από αυτή που αναμένουμε. Στις περιπτώσεις αυτές συμπεριλαμβάνονται και οι περιπτώσεις που το πρόγραμμα δεν τερματίζει ή δεν φτάνει στη γραμμή που μας ενδιαφέρει λόγω οποιουδήποτε λάθους κατά την εκτέλεση (runtime error). Στη διπλωματική αυτή εργασία θα επικεντρωθούμε στον εντοπισμό σφάλματος σε αντικειμενοστραφή (object-oriented) προγράμματα γραμμένα σε γλώσσα Java.

**Προτεινόμενη λύση**

Ο **δυναμικός τεμαχισμός προγράμματος** μας προσφέρει τη δυνατότητα να εντοπίσουμε τις γραμμές εκείνες του προγράμματος οι οποίες επηρεάζουν το αποτέλεσμα της μεταβλητής του ενδιαφέροντός μας για συγκεκριμένη εκτέλεση του προγράμματος με συγκεκριμένες τιμές εισόδου (το οποίο ορίζεται ως κριτήριο). Στην περίπτωση μας, ο δυναμικός προγραμματισμός μπορεί να μας δώσει τις σχετικές γραμμές του προγράμματος για την εκτέλεση ή τις εκτελέσεις που δεν μας δίνουν το αναμενόμενο αποτέλεσμα. Παρόλο που οι τεχνικές δυναμικού προγραμματισμού μας δίνουν μόνο τις γραμμές εκείνες που επηρεάζουν την τιμή της μεταβλητής για τη συγκεκριμένη εκτέλεση και συνεπώς ο αριθμός των γραμμών είναι μειωμένος σε σχέση με αυτόν των τεχνικών στατικού προγραμματισμού, εντούτοις ο αριθμός αυτός είναι τις περισσότερες φορές αρκετά μεγάλος κάνοντας δύσκολο τον εντοπισμό της συγκεκριμένης γραμμής που περιέχει το σφάλμα.

Οι **τεχνικές ελέγχου με χρήση μετάλλαξης (mutation testing)** εφαρμόζουν διάφορους τελεστές μετάλλαξης (mutant operators) εισάγοντας λάθη που κάνουν συχνά οι προγραμματιστές σε ένα σωστό πρόγραμμα. Δημιουργείται έτσι ένα μεγάλο σύνολο από λανθασμένα προγράμματα, τα ονομαζόμενα mutants, κάθε ένα από τα οποία περιέχει ένα συγκεκριμένο λάθος. Στη συνέχεια κάθε ένα από τα λανθασμένα προγράμματα εκτελείται με ένα σύνολο από σενάρια ελέγχου (test cases) με σκοπό τη μέτρηση της αποτελεσματικότητας του συνόλου αυτού με βάση την ικανότητα του να εντοπίσει τα λάθη.

Ένας τρόπος για εντοπισμό σφάλματος είναι η εφαρμογή αλλαγών / αντικαταστάσεων στον κώδικα του προγράμματος με σκοπό να αλλαχτεί η ροή του και να μας δώσει το επιθυμητό αποτέλεσμα. Αξιοποιώντας την πληροφορία που μας δίνει ο δυναμικός τεμαχισμός προγράμματος προσπαθούμε να αλλάξουμε τη ροή του εσφαλμένου προγράμματος εφαρμόζοντας αντικαταστάσεις **όχι σε ολόκληρο το πρόγραμμα αλλά μόνο στις συγκεκριμένες γραμμές που περιέχονται στο τεμάχιο της λανθασμένης εκτέλεσης του προγράμματός μας** όπως μας το έχει δώσει ο αλγόριθμος δυναμικού τεμαχισμού. Για την δημιουργία των αντικαταστάσεων που θα εφαρμόσουμε σε κάθε

μια από τις εν λόγω γραμμές χρησιμοποιούμε τους τελεστές μετάλλαξης του mutation testing και συγκεκριμένα τους τελεστές μετάλλαξης που υλοποιεί το εργαλείο muJava [OMK05] για προγράμματα Java. **Η αξιοποίηση των τελεστών μετάλλαξης του mutation testing μας βοηθά όχι μόνο στον εντοπισμό της γραμμής που περιέχει το λάθος αλλά και στην εύρεση του συγκεκριμένου λάθους της γραμμής και διόρθωσής του.**

Για κάθε γραμμή που περιέχεται στο τεμάχιο προγράμματος υπάρχουν πολλές αντικαταστάσεις ανάλογα με το είδος της δήλωσης. Η εύρεση της γραμμής ή γραμμών του κώδικα που περιέχουν το λάθος και της σωστής αντικατάστασης αποτελεί ένα πρόβλημα με πολύ μεγάλο αριθμό λύσεων. Χρησιμοποιώντας **γενετικούς αλγόριθμους** ανάγουμε το πρόβλημα αυτό σε πρόβλημα αναζήτησης.

Θα μπορούσαν φυσικά να χρησιμοποιηθούν και άλλες μέθοδοι αναζήτησης όπως Γενετικός Προγραμματισμός και Νευρωνικά Δίκτυα. Ωστόσο επιλέχθηκε η χρήση γενετικών αλγορίθμων μιας και έχουν δοκιμαστεί για τον έλεγχο συστημάτων λογισμικού [SASAK09]. Άρα λοιπόν λόγω προηγούμενης πείρας προτιμήθηκε αυτή η τεχνική.

## 7.2 Αναπαράσταση προβλήματος ως πρόβλημα γενετικών αλγορίθμων

**Κωδικοποίηση**

Στόχος μας είναι να βρούμε ποια ή ποιες από τις γραμμές που περιέχονται στο τεμάχιο του προγράμματος μας μπορεί να περιέχει το λάθος και επιπλέον ποια αντικατάσταση μπορεί να διορθώσει το λάθος.

Κάθε πιθανή λύση στο πρόβλημα μας αναπαριστάται ως ένα χρωματόσωμα (chromosome) με μέγεθος N, όπου N είναι ο αριθμός των γραμμών που περιέχονται στο τεμάχιο προγράμματος. Κάθε γραμμή του τεμαχίου αναπαριστάται από ένα γονίδιο (gene) το οποίο μπορεί να πάρει τιμές στο διάστημα **0..K.**

Κ είναι ο αριθμός των αντικαταστάσεων που έχουν δημιουργηθεί από τους τελεστές μετάλλαξης (mutant operators) για τη συγκεκριμένη γραμμή. Εφόσον ο αριθμός των αντικαταστάσεων που μπορεί να δημιουργηθεί για κάθε γραμμή είναι διαφορετικός, το διάστημα τιμών που μπορεί να πάρει κάθε γονίδιο είναι επίσης διαφορετικό.

Κάθε γονίδιο με τιμή μεγαλύτερη του 0 προτείνει ότι η αντίστοιχη γραμμή περιέχει το λάθος. Επιπλέον ο συγκεκριμένος αριθμός υποδηλώνει την αντικατάσταση που διορθώνει το λάθος. Πιο κάτω βλέπουμε ένα δείγμα πέντε πιθανών λύσεων (χρωματοσωμάτων):

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 4 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 2 | 5 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 3 | 3 | 4 | 4 |

Το πρώτο χρωματόσωμα προτείνει ότι το λάθος βρίσκεται στη $2^{η}$ γραμμή του τεμαχίου και ότι η $2^{η}$ αντικατάσταση που έχει δημιουργηθεί για τη γραμμή αυτή μπορεί να διορθώσει το λάθος. Το δεύτερο χρωματόσωμα προτείνει δύο γραμμές που μπορεί να περιέχουν το λάθος, την $3^{η}$ και την $6^{η}$, προτείνοντας επίσης την $4^{η}$ και $1^{η}$ αντικατάσταση αντίστοιχα για τις γραμμές αυτές.

**Χώρος αναζήτησης**

Με βάση την κωδικοποίησή μας, ο χώρος αναζήτησης του γενετικού αλγόριθμου για ένα τεμάχιο προγράμματος με μέγεθος **N** είναι:

$$A_1+1 * A_2+1 * A_3+1 * \ldots * A_N +1$$

όπου $A_x$ ο αριθμός των αντικαταστάσεων για την **x** γραμμή του τεμαχίου. Προσθέτουμε ένα σε αυτούς τους αριθμούς γιατί μια επιπρόσθετη επιλογή είναι και το 0.

**Συνάρτηση Καταλληλότητας**

Για να κατευθύνουμε τον γενετικό αλγόριθμο να βρει την καλύτερη λύση, αναθέτουμε στην συνάρτηση καταλληλότητας (fitness function) του γενετικού αλγόριθμου να τρέξει δυναμικά το αλλαγμένο πρόγραμμα μετά την εφαρμογή της κάθε προτεινόμενης αντικατάστασης με ένα συγκεκριμένο αριθμό σεναρίων που εκτελούνταν από το πρόγραμμα μας με επιτυχία (successful test cases) και με ένα ή περισσότερα σενάρια που εκτελούνταν από το πρόγραμμα μας λανθασμένα (faulty test cases) πριν από την αντικατάσταση και να βαθμολογήσει τη συγκεκριμένη αντικατάσταση με βάση:

- *Τον αριθμό των επιτυχών σεναρίων που παραμένουν επιτυχή μετά την αντικατάσταση*

και

- *Τον αριθμό των λανθασμένων σεναρίων που μετατρέπονται σε επιτυχή μετά την αντικατάσταση*

Μια γραμμή έχει μεγαλύτερη πιθανότητα να περιέχει το λάθος όσο μεγαλύτερος είναι και ο βαθμός της «καλύτερης» αντικατάστασης αυτής της γραμμής. Μια συγκεκριμένη λύση προτείνει μια ή περισσότερες γραμμές που περιέχουν το λάθος. *Ο αριθμός των γραμμών που περιέχονται σε κάθε λύση επηρεάζει αντιστρόφως ανάλογα την βαθμολογία της λύσης αυτής*.

Συγκεκριμένα κάθε προτεινόμενη λύση (χρωματόσωμα) βαθμολογείται με την ακόλουθη φόρμουλα:

$$\frac{\sum_{n=1..Size}((SSn \times SSn_{score} \times SSn_{weight}) + (FSn \times FSn_{score} \times FSn_{weight}))}{Size \times Size_{weight}}$$

**SSn:** αριθμός επιτυχών σεναρίων που παραμένουν επιτυχή μετά την αντικατάσταση n

**$SSn_{score}$:** σταθερή βαθμολογία που δίνεται για κάθε ένα από τα πιο πάνω σενάρια. Έχουμε ορίσει την τιμή 10 σε αυτή τη σταθερά.

**$SSn_{weight}$:** βάρος που ορίζεται από το χρήστη για βαθμολόγηση των πιο πάνω σεναρίων

**FSn:** αριθμός ανεπιτυχών σεναρίων που μετατρέπονται σε επιτυχή μετά την αντικατάσταση n

**$FSn_{score}$:** σταθερή βαθμολογία που δίνεται για κάθε ένα από τα πιο πάνω σενάρια. Έχουμε ορίσει την τιμή 10 σε αυτή τη σταθερά.

**$FSn_{weight}$:** βάρος που ορίζεται από το χρήστη για βαθμολόγηση των πιο πάνω σεναρίων

$\sum_{n=1..Size}$**:** άθροισμα βαθμολογίας κάθε αντικατάστασης της προτεινόμενης λύσης.

**Size:** αριθμός γραμμών που περιέχονται στην προτεινόμενη λύση, δηλαδή αριθμός γονιδίων με τιμή μεγαλύτερη του μηδενός.

**$Size_{weight}$:** βάρος που ορίζεται από το χρήστη για βαθμολόγηση του αριθμού των γραμμών της προτεινόμενης λύσης

**Κριτήριο Τερματισμού**

Ο αλγόριθμος τερματίζει και επιστρέφει το χρωματόσωμα με την καλύτερη βαθμολογία στις ακόλουθες περιπτώσεις:

- Όταν συμπληρώσει τον μέγιστο αριθμό των γενεών που του έχουμε καθορίσει. Στην δική μας περίπτωση έχουμε θέση την τιμή `200` στη μεταβλητή αυτή.
- Όταν εντοπίσει χρωματόσωμα με τη μέγιστη δυνατή βαθμολογία. Στην περίπτωση μας αυτό επιτυγχάνεται αν ένα χρωματόσωμα προτείνει μια μόνο γραμμή λάθους με την κατάλληλη αντικατάσταση που μετατρέπει όλα τα ανεπιτυχή σενάρια σε επιτυχή και δεν αλλοιώνει το αποτέλεσμα των επιτυχών σεναρίων. Μαθηματικά η περίπτωση αυτή αναπαριστάται από την ακόλουθη φόρμουλα

$$\frac{(SS \times SS_{score} \times SS_{weight}) + (FS \times FS_{score} \times FS_{weight})}{1 \times Size_{weight}}$$

Όπου σε αυτή την περίπτωση το **SS** ισούται με το ν αριθμό των επιτυχών σεναρίων που έχουν δοθεί σαν είσοδος και το **FS** με τον αριθμό των ανεπιτυχών σεναρίων που έχουν δοθεί σαν είσοδος

- Για τις τελευταίες X γενεές η βαθμολογία του καλύτερου χρωματοσώματος της γενεάς μειώνεται σε σύγκριση με αυτό της προηγούμενης γενεάς και ο αριθμός X αντιστοιχεί στο 25% των συνολικών γενεών που έχουμε ορίσει στον αλγόριθμο. Στην περίπτωση μας X=25% * 200= 50.

## 7.3 Περιγραφή αλγορίθμου / μεθοδολογίας

*Παρακάτω παρουσιάζεται αναλυτικά η μεθοδολογία που προτείνουμε για εντοπισμό σφάλματος σε προγράμματα Java. Αρχικά παρουσιάζεται ο αλγόριθμος υπό μορφή ψευδοκώδικα και στη συνέχεια ακολουθεί επεξήγηση του αλγορίθμου αλλά και της μεθοδολογίας γενικότερα.*

***Ψευδοκώδικας***

**GA Based Fault Localization algorithm**

```
//Create “normalized” code and generate mutants
Inputs: (a) location of faulty source code
        (b) mutant operators to be used
Call muJava libraries to “normalized” code and generate mutants


//Create program dynamic slice
Inputs: (a) Slicing criterion file based on normalized code
        (b) Faulty test case input values
        (c) “Normalized” code
Run JSlice tool on normalized code to create dynamic slice file for
faulty test case based on criterion

//Locate fault
Inputs:
       (a) location of normalized faulty code
       (b) dynamic slice file
       (c) Java filename and line no of criterion
       (d) Test cases filename
       (e) Weights for scoring (i) faulty to successful test cases
          (ii) successful to successful test cases and (iii) size of
          final slice

Read test cases file
```

```
Read dynamic slice file
For each slice line {
      Read actual statement from source file
      Read details of mutants/substitutions generated for statement
}


//Genetic Algorithm
//Initialize Genetic Algorithm parameners
Calculate maxFitness=
((successfullTestCasesNo*successToSuccessWeight*successToSuccessScore)
                          +
(failedTestCasesNo*failedToSuccessWeight* failedToSuccessScore))
                       /finalSliceSizeWeight;

For each slice statement
{
 Retrieve noOfSatatmentSubstitutions
 Set range of corresponding gene values
   minValue=0
   maxValue=noOfStatementSubstitutions
}
Create initial population chromosomes
Set fitness function
while ( evolutions<maxEvolutions ){
      evolve; //create new generation
      get CurrentMaxFitness
      if (currentMaxFitness>=maxFitness){
            break; //stop evolutions, fittest chromosome found
      }
      If currentMaxFitness<previousEvolutionMaxFitness{
            decreasingEvolutionsNo= decreasingEvolutionsNo+1;
      }
      If (decreasingEvolutionsNo= 25%* maxEvolutions){
            Stop; //fitness value is continuously decreasing
                  //in last generations
      }
}

//Print fittest chromosome  - final slice statements
Get fittest chromosome
For each gene of fittest chromosome {
      If (gene value > 0){
            Retrieve corresponding slice statement object
            Print original statement
            Retrieve substitution indicated form gene value from
            statement objects substitutions
            Print proposed substitution
      }
}
```

**Fitness Function algorithm**

```
//Call evaluate function for scoring current chromosome
```

```
chromosomeScore=0
For each gene of chromosome{
 finalSliceSize=0
 If (gene value > 0){
      finalSliceSize= finalSliceSize + 1;
      //calculate current statement substitution score
      Retrieve corresponding slice statement and details of
      substitution proposed by gene value
      If (substitutionstatement has already a score){
        currentSubstitutionScore=retrieve current score
        chromosomeScore=chromosomeScore+currentSubstitutionScore
      }else{
        //apply substitution and run test cases to score it
        currentSubstitutionScore=0;

        Copy normalized program source code to working dir
        Retrieve path of mutant file created for current
        substitution
        Replace corresponding source(java) file with mutant file
        in working dir
        Compile program source code
        For each test case object{
            Retrieve test case type, input
            values and expected
            Add an assert statement with expected output condition in
            java file of slicing criterion after criterion line no
            Run program with input values
            If (assert statement evaluates to true){
              If (test case type was successful){
               currentSubstitutionScore=
                  currentSubstitutionScore+
                  (successToSuccessWeight*successToSuccessScore)
              }else if (test case type was faulty){
                currentSubstitutionScore=
                  currentSubstitutionScore+
                  (failedToSuccessWeight*failedToSuccessScore)
              }
            }
         }//end for
      }
```

```
            chromosomeScore= chromosomeScore+ currentSubstitutionScore
        } //end for
        chromosomeScore=chromosomeScore /
                (finalSliceSize * finalSliceSizeWeight)
```

***Περιγραφή Αλγορίθμου / μεθοδολογίας:***

Αρχικά ο αλγόριθμος ζητά από το χρήστη να καθορίσει το φάκελο στον οποίο βρίσκονται τα αρχεία του Java προγράμματος. Ο αλγόριθμος μπορεί να δεχτεί πρόγραμμα με πολλά αρχεία java τα οποία μπορεί να ανήκουν και σε πακέτα (packages). Στην περίπτωση αυτή τα αρχεία πρέπει να βρίσκονται στους κατάλληλους υποφακέλους σύμφωνα με την μορφή των πακέτων.

Στη συνέχεια ο χρήστης μπορεί να καθορίσει τους τελεστές μετάλλαξης (mutant operators) που θα εφαρμοστούν στο πρόγραμμα. Οι τελεστές που χρησιμοποιούμε είναι αυτοί που υλοποιούνται από το εργαλείο muJava, ωστόσο έχουμε εξαιρέσει ένα μικρό αριθμό από αυτούς τους τελεστές που δεν εμπίπτουν στο δικό μας πλαίσιο εργασίας, μιας και ο στόχος εφαρμογής των τελεστών του muJava εργαλείου και γενικότερα του mutation testing δεν είναι ο ίδιος με το δικό μας στόχο. Συγκεκριμένα χρησιμοποιούμε τους πιο κάτω τελεστές (βλέπε 6.3 για επεξήγηση τελεστών):

| **Τελεστές σε επίπεδο μεθόδου** | **Τελεστές σε επίπεδο κλάσης** |
|---|---|
| **AOR** - Arithmetic Operator Replacement | **IHD** - Hiding variable deletion |
| **AOI** - Arithmetic Operator Insertion | **IOP** - Overriding method calling position change |
| **AOD** - Arithmetic Operator Deletion | **ISI** - super keyword insertion |
| **ROR** - Relational Operator Replacement | **ISD** - super keyword deletion |
| **COR** - Conditional Operator Replacement | **IPC** - Explicit call to a parent's constructor deletion |
| **COI** - Conditional Operator Insertion | **PNC** - new method call with child class type |
| **COD** - Conditional Operator Deletion | **PMD** - Member variable declaration with parent class type |
| **SOR** - Shift Operator Replacement | **PPD** - Parameter variable declaration with child class type |
| **LOR** - Logical Operator Replacement | |
| **LOI** - Logical Operator Insertion | |
| **LOD** - Logical Operator Deletion | |

| | |
|---|---|
| | **PCI** - Type cast operator insertion<br>Polymorphism PCC Cast type change<br>**PCD** - Type cast operator deletion<br>**PCC** - Cast type change<br>**PRV** - Reference assignment with other comparable variable<br>**OAC** - Arguments of overloading method call change<br>**JTI** - this keyword insertion<br>**JTD** - this keyword deletion<br>**JSI** - static modifier insertion<br>**JSD** - static modifier deletion<br>**JID** - Member variable initialization deletion<br>**EOA** - Reference assignment and content assignment replacement<br>**EOC** - Reference comparison and content comparison replacement<br>**EAM** - Acessor method change<br>**EMM** - Modifier method change |

Από τους πιο πάνω τελεστές ο χρήστης μπορεί να επιλέξει ποιούς από αυτούς θα χρησιμοποιήσει.

Στη συνέχεια χρησιμοποιούνται κλάσεις του εργαλείου muJava για παραγωγή των διαφόρων μεταλλάξεων (mutants) του προγράμματος μας εφαρμόζοντας τους τελεστές μετάλλαξης που έχει επιλέξει ο χρήστης. Πριν γίνει όμως αυτό, το εργαλείο muJava κάνει parse και αναλύει τις δηλώσεις του αρχικού προγράμματος και στη συνέχεια δημιουργεί ένα αντίγραφο του κώδικα σε «κανονικοποιημένη» μορφή. Η «κανονικοποιημένη» μορφή είναι ταυτόσημη με την αρχική μορφή κώδικα μόνο που αφαιρούνται αχρείαστα κενά στον κώδικα και σχόλια. Επίσης στους ορισμούς μεταβλητών τύπου αντικειμένου προστίθενται στον τύπο του αντικειμένου τυχόν πακέτα στα οποία ανήκει π.χ. το String αντικαθιστάται με java.lang.String.

Πάνω στην «κανονικοποιημένη» μορφή προγράμματος το εργαλείο εφαρμόζει τους τελεστές μετάλλαξης. Για κάθε τελεστή μετάλλαξης δημιουργούνται ένα ή περισσότερα

αντίγραφα του java αρχείου που εφαρμόστηκε η μετάλλαξη π.χ. ο αριθμητικός τελεστής AOR εφαρμόζεται σε όλες τις δηλώσεις του κώδικα που περιέχουν δυαδικό αριθμητικό τελεστή. Για κάθε τέτοια δήλωση δημιουργούνται πολλά αρχεία αφού αντικαθιστά π.χ. το + με -, *, /, %. Κάθε αντίγραφο φυλάγεται σε ξεχωριστό υποφάκελο με ονομασία AOR_1, AOR_2, AOR_3 κλπ. Παράλληλα δημιουργείται και ένα αρχείο log για τους τελεστές που εφαρμόστηκαν σε κάθε γραμμή.

Όταν τελειώσει η παραγωγή των μεταλλάξεων του κώδικα θα πρέπει ο χρήστης να δηλώσει το κριτήριο βάσει του οποίου θα δημιουργηθεί το δυναμικό τεμάχιο προγράμματος δηλαδή να δηλώσει το αρχείο και τον αριθμό γραμμής στον οποίο παρατηρείται η λανθασμένη έξοδος του προγράμματος ή η λανθασμένη τιμή μεταβλητής. Ο ορισμός του αριθμού γραμμής πρέπει να γίνει με βάση τον αριθμό γραμμών της «κανονικοποιημένης» μορφής αφού η αρίθμηση των γραμμών αλλοιώνεται. Ο «κανονικοποιημένος» κώδικας αντιγράφεται στο προηγούμενο βήμα του αλγορίθμου μας σε ένα υποφάκελο του αρχικού μας προγράμματος με ονομασία «normalized».

Για παραγωγή του δυναμικού τεμαχίου προγράμματος χρησιμοποιούμε το εργαλείο JSlice [WR04, JS08]. Με το εργαλείο αυτό τρέχουμε τον «κανονικοποιημένο» κώδικα μας με τις τιμές εισόδου για τις οποίες παρατηρήσαμε το λάθος στον κώδικα (faulty test case) και το κριτήριο υπό μορφή αρχείου που περιέχει το όνομα του αρχείου και τον αριθμό γραμμής του κριτηρίου μας. Το εργαλείο παράγει το τεμάχιο προγράμματος στο αρχείο εξόδου, για κάθε γραμμή του τεμαχίου δίνεται ο αριθμός της γραμμής και η ονομασία του αρχείου του κώδικα μας που περιέχεται η γραμμή αυτή.

Στη συνέχεια ο χρήστης τρέχει τον αλγόριθμό μας για τον εντοπισμό του λάθους δίνοντας σαν είσοδο τα ακόλουθα:

- Τοποθεσία (φάκελος) «κανονικοποιημένου» κώδικα.
- Το αρχείο του τεμαχίου προγράμματος όπως έχει δημιουργηθεί από το JSlice.
- Το όνομα του αρχείου του προγράμματος που περιέχει το κριτήριο που έχουμε ορίσει για το δυναμικό τεμαχισμό όπως και ο αριθμός της γραμμής του κριτηρίου.

- Το αρχείο με τα σενάρια μας. Συγκεκριμένα το αρχείο πρέπει να περιέχει ένα σύνολο σεναρίων (τιμών εισόδου) που εκτελούνται από το πρόγραμμα μας με επιτυχία (successful test cases) και ένα σύνολο σεναρίων που εκτελούνται από το πρόγραμμα μας λανθασμένα (faulty test cases). Για κάθε σενάριο δίνονται:
    - Ο τύπος σεναρίου (Επιτυχές/Λανθασμένο).
    - Οι τιμές εισόδου.
    - Το αναμενόμενο αποτέλεσμα της μεταβλητής που ορίζεται από το κριτήριο μας με βάση τις τιμές αυτές. Σημειώνεται ότι για τα λανθασμένα σενάρια δίνεται το αναμενόμενο αποτέλεσμα. Το αποτέλεσμα δηλαδή που θα είχε το πρόγραμμα μας αν ήταν σωστό με βάση τις τιμές αυτές και όχι το αποτέλεσμα που δίνει το λανθασμένο πρόγραμμά μας.

      *Το αναμενόμενο αποτέλεσμα ορίζεται από το χρήστη υπό τη μορφή μιας δήλωσης σε γλώσσα java που θα πρέπει να αποτιμάται σε true ή false σύμφωνα με τα συμφραζόμενα του προγράμματος μας.*

      *π.χ* `x.getMax(a,b,c)==a && y.getAverage(d,e)=3`

      *Αυτό μας δίνει την ευελιξία να εκφράσουμε οποιοδήποτε αποτέλεσμα και συγχρόνως να ορίσουμε σαν κριτήριο οποιαδήποτε έκφραση και όχι απλώς μια μεταβλητή που να επιστρέφει μια σταθερά π.χ.* `χ==3`.

      Πιο κάτω δίνεται ένα παράδειγμα αρχείου σεναρίων:

```
com\mindprod\creditcard\ValidateCreditCard.java 1
F;379999999999999;vendor_name.equals("AMEX")
S;4000000000000;vendor_name.equals("Visa")
S;4999999999999;vendor_name.equals("Visa")
S;4999999999998;vendor_name.equals("Visa")
S;0;vendor_name.equals("Error: not enough digits")
F;6011222233334444;vendor_name.equals("Discover")
```

      Στην πρώτη γραμμή του αρχείου των σεναρίων δίνεται επίσης το όνομα του κυρίως αρχείου του προγράμματος μας (που περιέχει την main) και ο αριθμός των παραμέτρων που λαμβάνει ως είσοδο. Το συγκεκριμένο

πρόγραμμα παίρνει σαν είσοδο ένα αριθμό. Ο τύπος σεναρίου ορίζεται με τα γράμματα «S» και «F». Το σύμβολο «;» χρησιμοποιείται ως διαχωριστικός χαρακτήρας.

- Τρεις δεκαδικοί αριθμοί που ορίζουν τα βάρη που θα δοθούν κατά τη βαθμολόγηση κάθε προτεινόμενης αντικατάστασης/μετάλλαξης. Συγκεκριμένα ο χρήστης ορίζει τα πιο κάτω βάρη:
  - `successToSuccessWeight:` βάρος βαθμολόγησης κάθε επιτυχούς σεναρίου που παραμένει επιτυχές μετά την αντικατάσταση.
  - `failedToSuccessWeight:` βάρος βαθμολόγησης κάθε ανεπιτυχούς σεναρίου που μετατρέπεται σε επιτυχές μετά την αντικατάσταση.
  - `finalSliceSizeWeigh:` βάρος βαθμολόγησης του αριθμού γραμμών που περιέχονται στο τελικό τεμάχιο προγράμματος που προτείνεται ως λύση. Δηλαδή αριθμός γραμμών που προτείνουμε ότι περιέχουν το λάθος.

Το πρώτο βήμα στη διαδικασία εντοπισμού του λάθους είναι η ανάγνωση του αρχείου σεναρίων ελέγχου. Στη συνέχεια διαβάζεται το αρχείο του τεμαχίου προγράμματος. Για κάθε γραμμή που περιέχεται σε αυτό διαβάζουμε από το αντίστοιχο java αρχείο του προγράμματος τη δήλωση που περιέχει η γραμμή αυτή αφού το αρχείο του τεμαχίου μας υποδεικνύει απλώς το αριθμό της γραμμής. Επίσης βρίσκουμε για κάθε γραμμή τις λεπτομέρειες τυχόν μεταλλάξεων που έχουν δημιουργηθεί για αυτή όπως ο τελεστής μετάλλαξης που εφαρμόστηκε, η τοποθεσία φύλαξης του μεταλλαγμένου αρχείου και η ακριβής αλλαγή κώδικα που έγινε στη μετάλλαξη αυτή (αντικατάσταση). Οι λεπτομέρειες αυτές εξάγονται από την ανάλυση του log αρχείου που δημιουργεί το muJava.

Το επόμενο βήμα είναι η αρχικοποίηση των παραμέτρων του γενετικού αλγόριθμου για να ξεκινήσει η διαδικασία αναζήτησης. Αρχικά υπολογίζεται η μέγιστη βαθμολογία (maxFitness) που μπορεί να πάρει μια λύση στο πρόβλημα μας με βάση τον συνολικό αριθμό των επιτυχών σεναρίων, τον αριθμό των ανεπιτυχών σεναρίων, τα βάρη που ορίστηκαν από το χρήστη, τις σταθερές βαθμολόγησης μας και τον

αριθμό γραμμών της λύσης, χρησιμοποιώντας τη φόρμουλα που παρουσιάστηκε πιο πάνω.

Μετά, ορίζουμε το διάστημα τιμών πο υ μπορεί να πάρει κάθε γονίδιο το υ χρωματοσώματος μας με βάση τον αριθμό μεταλλάξεων N της γραμμής που αντιπροσωπεύει. Όπως έχουμε αναφέρει και ποιο πάνω το διάστημα αυτό είναι 0 μέχρι N.

Στη συνέχεια αφού δημιουργηθεί ο αρχικός πληθυσμός χρωματοσωμάτων αρχίζει η διαδικασία αναπαραγωγής και αναζήτησης της λύσης που είναι ίσος με 50.
Ακολούθως ορίζεται η συνάρτηση καταλληλότητας που θα χρησιμοποιηθεί από το γενετικό αλγόριθμο, η συνάρτηση καταλληλότητας περιγράφεται πιο κάτω.

Όπως έχουμε περιγράψει και στο 7.3, ο γενετικός αλγόριθμος σταματά αν φτάσουμε το μέγιστο αριθμό γενεών που έχουμε ορίσει (στην περίπτωση μας έχουμε δώσει τον αριθμό 200 σε αυτή τη μεταβλητή), αν σε μια γενεά βρούμε χρωματόσωμα με τη μέγιστη δυνατή βαθμολογία ή αν η καλύτερη βαθμολογία κάθε γενεάς μειώνεται σε σύγκριση με αυτή της προηγούμενης για αριθμό γενεών που αντιστοιχεί σο 25% του μέγιστου αριθμού γενεών μας.

Μετά τον τερματισμό του αλγορίθμου παρουσιάζεται η καλύτερη λύση στο χρήστη ως ακολούθως: Για κάθε γονίδιο του χρωματοσώματος που έχει τιμή μεγαλύτερη του 0, τυπώνεται η αρχική δήλωση της αντίστοιχης γραμμής κώδικα του προγράμματος μας όπως και ο αριθμός της γραμμής και η ονομασία του αρχείου στο οποίο βρίσκεται. Ανάλογα επίσης με την τιμή του γονιδίου προτείνεται η αντικατάσταση που διορθώνει το λάθος στη συγκεκριμένη γραμμή ανακτώντας τις πληροφορίες της αντίστοιχης μετάλλαξης της γραμμής. Ένα παράδειγμα λύσης παρουσιάζεται πιο κάτω:

```
org\testfiles\TriangClassification.java Line No:30
if (i + j == k || j + k <= i || i + k <= j) {

Proposed substitution:  i + j == k  =>   i + j <= k
```

**Συνάρτηση καταλληλότητας (fitness function)**

Η συνάρτηση καταλληλότητας είναι αυτή που βαθμολογεί κάθε πιθανή λύση στο πρόβλημά μας (χρωματόσωμα) και κατευθύνει με αυτό τον τρόπο τον γενετικό αλγόριθμο να βρει την καλύτερη λύση.

Για κάθε γονίδιο του χρωματοσώματος με τιμή μεγαλύτερη του 0 υπολογίζουμε τη βαθμολογία της προτεινόμενης αντικατάστασης. Κοιτάζουμε πρώτα αν έχουμε ήδη βαθμολογήσει τη συγκεκριμένη αντικατάσταση για τη συγκεκριμένη γραμμή κώδικα κατά τη διάρκεια βαθμολόγησης ενός προηγούμενου χρωματοσώματος. Αν δεν υπάρχει βαθμολογία, την υπολογίζουμε ως ακολούθως:

Αντιγράφουμε τα αρχεία του «κανονικοποιημένου» προγράμματος μας σε ένα προσωρινό φάκελο για να δουλέψουμε με αυτά. Αντικαθιστούμε το αρχείο στο οποίο εφαρμόστηκε η συγκεκριμένη μετάλλαξη/αντικατάσταση με το java αρχείο που έχει δημιουργηθεί από το muJava. Στη συνέχεια μεταγλωττίζουμε και τρέχουμε το πρόγραμμα μας με όλα τα σενάρια ελέγχου.

Για κάθε σενάριο ελέγχου ανακτούμε τις τιμές εισόδου του σεναρίου και τη δήλωση του αναμενόμενου αποτελέσματος όπως μας έχει δοθεί από το χρήστη και προσθέτουμε ένα assert statement με τη συγκεκριμένη δήλωση ακριβώς μετά τη γραμμή του κριτηρίου μας στο ανάλογο αρχείο. π.χ αν η δήλωση ήταν `x.getMax(a,b)==a` προσθέτουμε τη γραμμή: **`assert (x.getMax(a,b)==a)`**
Μεταγλωττίζουμε ξανά το αλλαγμένο αρχείο και τρέχουμε το πρόγραμμα μας με τις συγκεκριμένες τιμές εισόδου. Αν το assert statement αποτιμηθεί σε ορθό (true) τότε:

- αν το σενάριο ελέγχου μας ήταν επιτυχές τότε προσθέτουμε στην τρέχουσα βαθμολογία της συγκεκριμένης αντικατάστασης `successToSuccessWeight*successToSuccessScore`, στην υλοποίηση μας έχουμε δώσει το αριθμό 10 στη σταθερά `successToSuccessScore)`.

- αν το σενάριο ελέγχου μας ήταν ανεπιτυχές τότε προσθέτουμε στην τρέχουσα βαθμολογία της συγκεκριμένης αντικατάστασης

`failedToSuccessWeight*failedToSuccessScore`, στην υλοποίηση μας έχουμε επίσης δώσει το αριθμό 10 στη σταθερά `failedToSuccessScore`).

Όταν γίνει αυτό για όλα τα σενάρια, προσθέτουμε τη συνολική βαθμολογία της συγκεκριμένης αντικατάστασης στην τρέχουσα βαθμολογία του χρωματοσώματος.

Η βαθμολογία του χρωματοσώματος διαιρείται στο τέλος με την τιμή `finalSliceSize*finalSliceSizeWeight`, όπου `finalSliceSize` είναι ο αριθμός όλων των γονιδίων που προτείνουν κάποια αντικατάσταση (δηλαδή με τιμή > 0) και συνεπώς προτείνονται σαν γραμμές που περιέχουν το λάθος.

## 7.4 Υλοποίηση εργαλείου λογισμικού

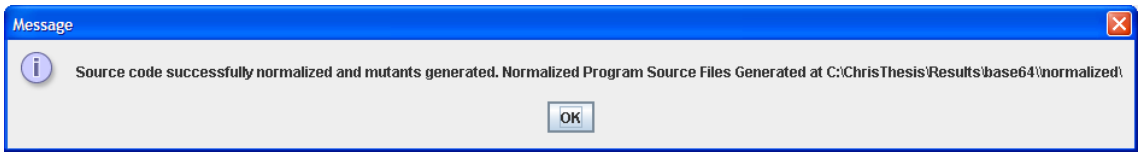
Με βάση τη μεθοδολογία που περιγράφουμε πιο πάνω έχουμε αναπτύξει ένα εργαλείο λογισμικού με σκοπό την αυτοματοποίηση του εντοπισμού σφάλματος αλλά και τη διόρθωση του. Στο υποκεφάλαιο αυτό παρουσιάζουμε τεχνικές λεπτομέρειες της υλοποίησής μας. Δίνονται επίσης πληροφορίες που αφορούν τη χρήση εξωτερικών εργαλείων που εμπλέκονται στη διαδικασία μας.

Για την ανάπτυξη του εργαλείου χρησιμοποιούμε τη γλώσσα java. Σαν περιβάλλον ανάπτυξης του εργαλείου μας έχουμε χρησιμοποιήσει το Eclipse SDK v3.4.2. Η διαπροσωπεία του εργαλείου χωρίζεται σε δύο μέρη, ένα για την **«κανονικοποίηση» του κώδικα και την παραγωγή των μεταλλάξεων** και ένα για τον **εντοπισμό του σφάλματος**. Η **παραγωγή του τεμαχίου του προγράμματος** γίνεται εξωτερικά με χρήση του εργαλείου JSlice.

**Κανονικοποίηση» κώδικα και παραγωγή μεταλλάξεων**

Όπως έχουμε αναφέρει η «κανονικοποίηση» και παραγωγή μεταλλάξεων γίνεται με κλήση κλάσεων του εργαλείου muJava. Συγκεκριμένα χρησιμοποιούνται οι κλάσεις `MutationSystem` για καθορισμό παραμέτρων του muJava όπως ο φάκελος από το οποίο το muJava θα διαβάσει τον αρχικό μας κώδικα., ο φάκελος που θα φυλάξει την «κανονικοποιημένη» μορφή του και ο φάκελος που θα φυλάξει τις μεταλλάξεις του κώδικα. Στη συνέχεια χρησιμοποιούνται οι κλάσεις `TraditionalMutantsGenerator` και `ClassMutantsGenerator` για παραγωγή των μεταλλάξεων σε επίπεδο μεθόδου και κλάσης αντίστοιχα. Σε κάθε μια από τις κλάσεις αυτές δίνουμε σαν παράμετρο τους αντίστοιχους τελεστές που έχει επιλέξει ο χρήστης να χρησιμοποιήσει.

Μετά από επιτυχή εκτέλεση της διαδικασίας αυτής υποδεικνύεται στο χρήστη ο φάκελος στον οποίο έχει παραχθεί η «κανονικοποιημένη» μορφή του κώδικα μας. Ο φάκελος αυτός βρίσκεται κάτω από το φάκελο του αρχικού μας προγράμματος με την ονομασία «normalized».




Το εργαλείο σε αυτό το μέρος της διαδικασίας όπως και στο επόμενο παράγει ένα λεπτομερές log αρχείο με τα βήματα που έχουν εκτελεστεί και τις σχετικές πληροφορίες. Για την παραγωγή του αρχείου χρησιμοποιείται το Apache Log4j [ALSP09]. Ένα παράδειγμα αρχείου για το πρώτο στάδιο της διαδικασίας φαίνεται πιο κάτω:

```
2009-12-23 00:22:15,815 [main] INFO  gafaultlocator.FaultLocator – Normalizing
and Generating MuJava Mutators in Progress...
2009-12-23 00:22:16,800 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Processing File...  com\mindprod\base64\Example.java
2009-12-23 00:22:16,815 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Generating Class Level Mutators...
2009-12-23 00:22:17,268 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Generating Method Level Mutators...
2009-12-23 00:22:17,628 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Processing File...  com\mindprod\base64\Base64.java
```

```
2009-12-23 00:22:17,628 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Generating Class Level Mutators...
2009-12-23 00:22:19,440 [main] INFO  gafaultlocator.FaultLocator - MuJava -
Generating Method Level Mutators...
2009-12-23 00:22:36,550 [main] INFO  gafaultlocator.FaultLocator - Normalized
Program Source Files Generated at C:\ChrisThesis\Results\base64\normalized
```

**Παραγωγή Τεμαχίου Προγράμματος**

Ο χρήστης σε αυτό το στάδιο καλείται να καθορίσει το κριτήριο με βάση το οποίο θα δημιουργηθεί το τεμάχιο προγράμματος. Ουσιαστικά θα πρέπει να καθορίσει το όνομα του αρχείου και τον αριθμό γραμμής του κριτηρίου σε ένα αρχείο με την μορφή που ορίζει το JSlice και με την ονομασία `_criterions`. Ένα παράδειγμα τέτοιου αρχείου φαίνεται πιο κάτω:

```
1
Example.java
22
Multiple
```

Το πιο πάνω αρχείο καθορίζει το κριτήριο μας στη γραμμή `22` του αρχείου `Example.java`. Υπάρχει δυνατότητα για ορισμό πολλαπλών κριτηρίων αλλά για τους σκοπούς της δικής μας εργασίας χρησιμοποιούμε μόνο ένα. Κατά συνέπεια η πρώτη και η τελευταία γραμμή του αρχείου κριτηρίων θα είναι πάντοτε όπως φαίνονται πιο πάνω. Στη συνέχεια θα πρέπει να τρέξουμε στο πρόγραμμα μας στο JSlice δίνοντας ως είσοδο το αρχείο κριτηρίου και τις τιμές εισόδου του αρχείου μας για τις οποίες έχουμε το λανθασμένο αποτέλεσμα με τον πιο κάτω τρόπο:

```
$slicingRoot/bin/java  -noclassgc -slicing
   -foreclipse  $appRoot/_criterions $dynamicSliceResult -classpath
   $slicingRoot/lib/Klasses.jar:$slicingRoot/lib/kjc.jar:$classRoot
   [Main Class of the Java program] <parameters>
```

όπου `$slicingRoot`: ο φάκελος που είναι εγκατεστημένο το JSlice,

`$appRoot`: ο φάκελος που βρίσκεται ο κώδικας του προγράμματος μας και το αρχείο κριτηρίου και

`$classRoot`: ο φάκελος που βρίσκονται οι μεταγλωττισμένα αρχεία (.class) του προγράμματος μας.

`$dynamicSliceResult:` ο φάκελος και το όνομα αρχείου στο οποίο θα δημιουργηθεί το τεμάχιο προγράμματος

Ένα παράδειγμα χρήσης της πιο πάνω εντολής είναι το ακόλουθο:

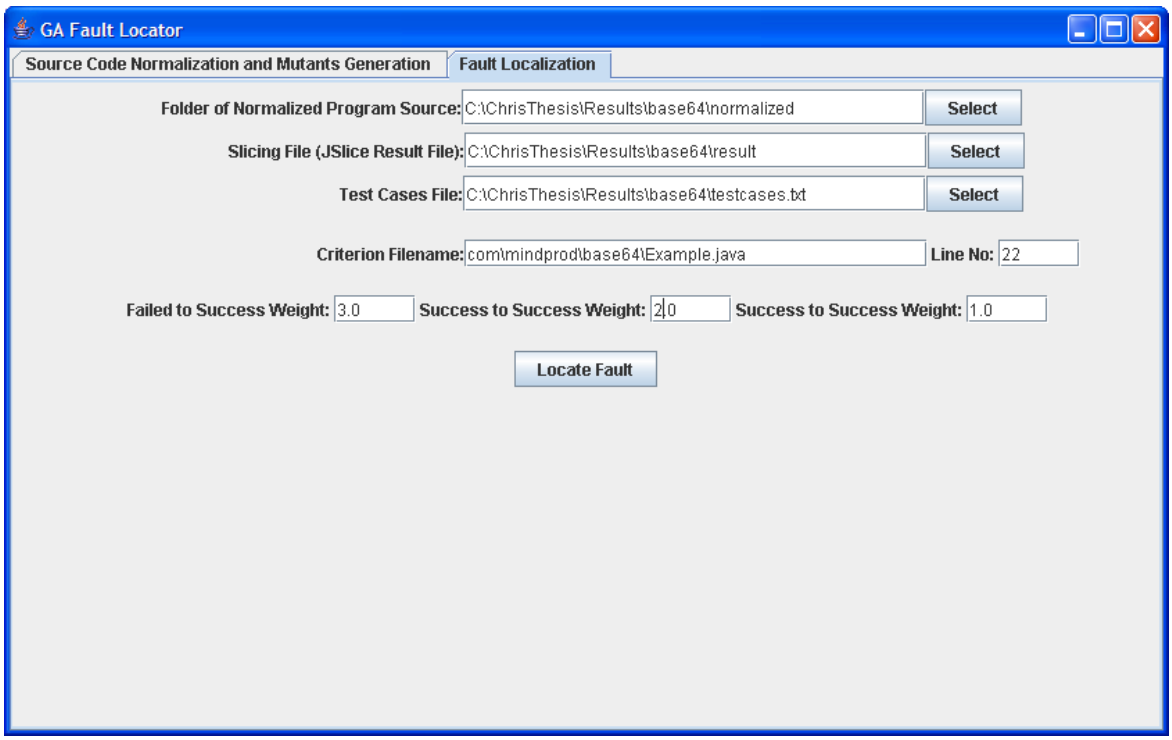
```
java -noclassgc -slicing -foreclipse /root/Slicing/base64/_criterions
/root/Slicing/base64/result -classpath
/usr/local/lib/Klasses.jar:/usr/local/lib/kjc.jar:/root/Slicing/base64
com.mindprod.base64.Example abc
```

Ένα τμήμα του αρχείου του τεμαχίου προγράμματος που παράγεται για το πιο πάνω παράδειγμα είναι το ακόλουθο:

```
.
.
Base64.java
227
Base64.java
231
Base64.java
232
Example.java
13
Example.java
14
Example.java
15
.
.
```

Όπως βλέπουμε πιο πάνω το αρχείο του τεμαχίου περιέχει για κάθε γραμμή κώδικα που ανήκει στο τεμάχιο το όνομα του java αρχείου και τον αριθμό της γραμμής. Να σημειωθεί ότι οι γραμμές δεν είναι κατηγοριοποιημένες κατά αρχείο ούτε ταξινομημένες κατά αριθμό γραμμής αλλά παρουσιάζονται με τη σειρά που έχουν εκτελεστεί από το πρόγραμμά μας. Τέλος στο όνομα του αρχείου java δεν περιέχεται η ονομασία τυχόν πακέτων στα οποία ανήκει.

**Εντοπισμός Σφάλματος.**




**Εικόνα 7.22 - Διαπροσωπεία Εντοπισμού Σφάλματος**

*Η κλάση που υλοποιεί τη διαπροσωπεία μας είναι η ακόλουθη:*

```
package gafaultlocator;

public class FaultLocatorGUI extends JFrame {

    public FaultLocatorGUI();

    public JPanel locateFaultPanel();

    public JPanel generateMutatorsNormalizePanel();
}
```

Η κύρια κλάση της υλοποίησης μας, η οποία και καλείται από τις μεθόδους της διαπροσωπείας είναι η `FaultLocator`:

```
package gafaultlocator;

public class FaultLocator {

  protected boolean normalizeAndGenerateMutators(
                            String[] traditionalOperators,
                            String[] classOperators);

  Protected void locateFault(String sliceFilePath,
                            String criterionFileName,
                            int criterionLineNo,
                            String testcasesPath,
                            double failedToSuccessWeight,
```

```
                              double successToFailedWeight,
                              double finalSliceSizeWeight);


  private void sortProgramSliceLines(String sliceFile);

  private void readTestCases(String testCasesFile);

  private void createSliceStatementObjects(String sliceFile,
                                           String srcRoot)´

  private void getMuJavaSubstitutions(SliceStatement sliceStmt,
                                      String muJavaResultFilePath,
                                      char resultFileType);

  public void setSrcRoot(String srcDir);

  public String getSrcRoot();

  public void setMainJavaFile(String mainJavaFile);

  public String getMainJavaFile();

  public Vector getLookupTable();

  public Vector getTestCases();
}
```

Η μέθοδος `normalizeAndGenerateMutators` είναι αυτή που καλείται στο πρώτο μέρος της διαδικασίας μας για την «κανονικοποίηση» και την παραγωγή μεταλλάξεων. Για εντοπισμό σφάλματος η κύρια μέθοδος είναι η `locateFault`. Αυτή καλεί πρώτα τη μέθοδο `copyDir` της κλάσης `CopyDir` που φαίνεται πιο κάτω για αντιγραφή του κώδικα του προγράμματος μας σε ένα φάκελο εργασίας. Η μέθοδος `copyDir` εκτός από αντιγραφή των java αρχείων, με βάση τους υποφακέλους που βρίσκεται το κάθε αρχείο, δημιουργεί και ένα πίνακα με το τυχόν πακέτο στο οποίο βρίσκεται κάθε αρχείο μιας και το JSlice δεν περιλαμβάνει αυτή την πληροφορία στο αρχείο του τεμαχίου προγράμματος, κάτι που θα χρειαστούμε στη συνέχεια.

Η ανάγνωση των σεναρίων ελέγχου που λαμβάνει χώρα στη συνέχεια υλοποιείται από τη μέθοδο `readTestCases`. Για κάθε σενάριο δημιουργείται ένα αντικείμενο της κλάσης `TestCase` που φαίνεται πιο κάτω.

Η μέθοδος `sortProgramSliceLines` χρησιμοποιείται για ταξινόμιση των γραμμών του τεμαχίου ανά αρχείο και αριθμό γραμμής για ευκολότερη επεξεργασία στη συνέχεια. Η `createSliceStatementObjects` διαβάζει τη δήλωση κάθε γραμμής του τεμαχίου από

το αντίστοιχο αρχείο του προγράμματος μας και δημιουργεί για κάθε γραμμή ένα αντικείμενο της κλάσης `SliceStatement`. Για κάθε γραμμή καλείται επίσης η μέθοδος `getMuJavaSubstitutions` η οποία ανακτά τις λεπτομέρειες τυχόν μεταλλάξεων της γραμμής από το log αρχείο του muJava. Για κάθε τέτοια μετάλλαξη δημιουργείται ένα αντικείμενο της κλάσης `SubstitutionStatement`. Όλα τα αντικείμενα αυτά προστίθονται στο `substitutions Vector` του αντικειμένου `SliceStatement` που έχουμε δημιουργήσει για τη γραμμή.

Αφού δημιουργηθούν τα απαραίτητα αντικείμενα καλείται η κλάση `FLGeneticAlgorithm` για εκτέλεση του γενετικού αλγορίθμου με σκοπό τον εντοπισμό του λάθους.

```
package gafaultlocator;

public class CopyDir {

      public void copyDir(String srcDir, String dstDir,
                          String extension) throws IOException;

      public void copyFile(String srcFile, String dstFile,
          boolean addToJavaSrcPaths) throws IOException;

      public Hashtable getSrcFilePaths();

      public static boolean deleteDir(File dir);
}
```

```
package gafaultlocator;

public class SliceStatement {

      private String srcFile;
      private String originalStatement;
      private int lineNo;
      private Vector substitutions;
      private boolean srcCopiedToWorkingDir;

      public SliceStatement(String srcFile, int lineNo,
                            String originalStatement);

      public void setSrcFile(String srcFile);

      public String getSrcFile();

      public void setLineNo(int lineNo);

      public int getLineNo();

      public void setOriginalStatement(String originalStatement);

      public String getOriginalStatement();

      public void setSubstitutions(Vector substitutions);
      public Vector getSubstitutions();

      public void SetSrcCopiedToWorkingDir(boolean value);

      public boolean getSrcCopiedToWorkingDir();
}
```

```
package gafaultlocator;

public class SubstitutionStatement {

      private String statement;
      private int statementLineNo
      private int score=-1;
      private String mutantOperator;
      private String substitutionDescr;
      private String srcFilePathAfterSubstitution;

      public void setStatement(String statement);

      public void setStatementLineNo(int statementLineNo);

      public void setScore(int score);

      public void setMutantOperator(String mutantOperator);

      public String getStatement();

      public int getStatementLineNo();

      public int getScore();

      public String getMutantOperator();

      public void setSubstitutionDescr(String substitutionDescr);

      public String getSubstitutionDescr();

      public void setSrcFilePathAfterSubstitution(String
                                          srcFilePathAfterSubstitution);
      public String getSrcFilePathAfterSubstitution();
      }
}
```

Η κλάση `FLGeneticAlgorithm` υλοποιεί το γενετικό μας αλγόριθμό. Για την υλοποίηση του γενετικού αλγορίθμου χρησιμοποιούμε το πακέτο γενετικών αλγορίθμων **JGAP** [JG08]. Αφού γίνει η αρχικοποίηση των διαφόρων παραμέτρων του αλγορίθμου όπως περιγράψαμε στο προηγούμενο υποκεφάλαιο αρχίζει η αναζήτηση χρησιμοποιώντας ως συνάρτηση καταλληλότητας την `evaluate` μέθοδο της κλάσης `FLFitnessFunction`.

```
package gafaultlocator;

public class FLGeneticAlgorithm {

      public FLGeneticAlgorithm(int chromosomeSize,
         Vector sliceStatements, Vector testCases,
         String programSourceRoot, String mainJavaFile,
```

```
          String criterionFileName, int criterionLineNo,
          double failedToSuccessWeight, double successToSuccessWeight,
          double finalSliceSizeWeight);
}
```

```
package gafaultlocator;

public class FLFitnessFunction extends FitnessFunction {

      public FLFitnessFunction(Vector sliceStatements, Vector
          testCases, String programSourceRoot, String mainJavaFile,
          String criterionFileName, int criterionLineNo,
          double failedToSuccessWeight, double successToSuccessWeight,
          double finalSliceSizeWeight);

      public void addOrReplaceAssertStatement(String srcFilePath,
          int lineno, String assertStatement);

      private boolean runTestCase(String srcRootDir,
          String mainFileName, String[] params,
          String changedJavaFile);

      public double evaluate(IChromosome a_subject);

      private int calculateStatementSubstitutionScore(
              int sliceStmtPosition, int geneValue);
}
```

*Η μέθοδος* `evaluate` *παίρνει σαν είσοδο ένα χρωματόσωμα που αντιπροσωπεύει μια πιθανή λύση στο πρόβλημα μας και αξιολογεί τη λύση ακολουθώντας τα βήματα που έχουμε περιγράψει στον αλγόριθμό μας. Η* `calculateStatementSubstitutionScore` *καλείται από την μέθοδο* `evaluate` *για κάθε γονίδιο με τιμή μεγαλύτερη του 0 για να βαθμολογήσει την αντικατάσταση που προτείνεται από το γονίδιο αυτό. Αν η συγκεκριμένη αντικατάσταση για αυτή τη γραμμή του τεμαχίου δεν έχει βαθμολογηθεί προηγούμενος και φυλαχτεί στο αντίστοιχο* `SubstitutionStatement` *αντικείμενο, η* `calculateStatementSubstitutionScore`*:*

- *αντιγράφει ξανά τον αρχικό μας κώδικα για κάθε σενάριο ελέγχου,*
- *βρίσκει το java αρχείο που δημιουργήθηκε από το muJava για τη συγκεκριμένη εγκατάσταση/μετάλλαξη και αντικαθιστά το αντίστοιχο αρχείο του προγράμματος μας και*
- *καλεί τη μέθοδο* `addOrReplaceAssertStatement` *για να προσθέσει μετά τη γραμμή του κριτηρίου μας την assert δήλωση με το αναμενόμενο αποτέλεσμα.*

*Η μέθοδος* **`runTestCase`** **είναι αυτή που στη συνέχεια**

- *μεταγλωττίζει ξανά τα αλλαγμένα java αρχεία του προγράμματος μας με χρήση της μεθόδου* `com.sun.tools.javac.Main.compile`
- *δημιουργεί ένα νέο* `ClassLoader` *ο οποίος ξαναφορτώνει δυναμικά της αλλαγμένες κλάσεις και*
- *τρέχει το πρόγραμμα μας καλώντας δυναμικά την main μέθοδο του με τις εισόδους του συγκεκριμένου σεναρίου ελέγχου χρησιμοποιώντας τις δυνατότητες του* `Java Reflection`*.*

Σε περίπτωση που μετά το κάλεσμα τις main λάβει AssertionError exception λόγω αποτίμησης της δήλωσης assert σε false ή λάβει οποιοδήποτε άλλα runtime exception τότε θεωρεί ότι το σενάριο εκτελέστηκε ανεπιτυχώς και δεν προσθέτει κάποιο βαθμό στη μέχρι στιγμής βαθμολογία της συγκεκριμένης εγκατάστασης. Σε αντίθετη περίπτωση προσθέτει στη βαθμολογία ανάλογα με το είδος του σεναρίου και με βάση τα αντίστοιχα βάρη βαθμολόγησης όπως περιγράψαμε προηγουμένως.

Ένα τμήμα του σχετικού κώδικα για δυναμική εκτέλεση και αποτίμηση του αποτελέσματος του προγράμματος παρατίθεται πιο κάτω:

```
Object classParameters[] = {params};
Class classParametersTypes[] ={classParameters[0].getClass()};
Method theMethod =

aClass.getDeclaredMethod("main",classParametersTypes);
try{
    theMethod.invoke(null, classParameters);
    successful=true;
}catch(InvocationTargetException e){
  if (e.getCause().toString().equals("java.lang.AssertionError")){
     //assertion exception thrown --> test case is false
      logger.debug("AssertionError");
         successful=false;
  }else{
        logger.debug("Runtime Exception on running test case");
        successful=false;
  }
}
```

Μετά από τη βαθμολόγηση του κάθε γονιδίου στο οποίο προτείνεται κάποια αντικατάσταση και συνεπώς προτείνεται από το συγκεκριμένο χρωματόσωμα ότι μπορεί να περιέχει το λάθος, η μέθοδος `evaluate` επιστρέφει τη συνολική βαθμολογία του χρωματοσώματος αφού πρώτα διαιρέσει με το συνολικό αριθμό τέτοιων γονιδίων πολλαπλασιασμένο με το αντίστοιχο βάρος που έχει καθοριστεί από το χρήστη.

*Η τελευταία κλάση της υλοποίησης μας είναι η `Constants` στην οποία ορίζουμε τις διάφορες σταθερές που χρησιμοποιεί το εργαλείο μας όπως οι σταθερές του γενετικού αλγορίθμου και το σύνολο των τελεστών μετάλλαξης που χρησιμοποιούμε.*

```
package gafaultlocator;

public final class Constants {

	public static final String FILE_SEPARATOR =
	                              System.getProperty("file.separator");
	public static final String WORKING_DIR= ".\\working";
	public static final String MUJAVA_HOME="C:\\MuJava";
	public static final int EVOLUTIONS =200;
	public static final int POPULATION_SIZE = 50;
	public static final double FITNESS_DECREASING_PERCENT =0.25;
	public static final int FAILED_TO_SUCCESS_SCORE = 10;
	public static final int SUCCESS_TO_FAILED_SCORE = 0;
	public static final int SUCCESS_TO_SUCCESS_SCORE = 10;
	public static final int FAILED_TO_FAILED_SCORE = 0;
	public static final String LOGGER_PROPERTIES_FILE =
	            "C:\\ChrisThesis\\GAFaultLocator\\logger.properties";

	public static final String DEFAULT_PROGRAM_SOURCES_FOLDER=
	                                    "C:\\ChrisThesis\\Results\\";
	public final static String[] class_ops=new String[] {
	   "IHD","ISI","ISD","IPC","PNC","PMD","PPD",,"PCI","PCD",
	    "PCC","PRV","OAC","JTI","JTD","JSI","JSD","JID",
	     "EOA","EOC","EAM","EMM"};

	public final static String[] traditional_ops=new String[] {
	     "AORB","AORS","AOIU","AOIS","AODU","AODS","ROR",
	     "COR","COD","COI","SOR","LOR","LOI","LOD","ASRS"};
}
```

Το εργαλείο μας κατά τη διάρκεια της διαδικασίας εντοπισμού και πρότασης διόρθωσης του σφάλματος δημιουργεί αναλυτικό αρχείο log με τα βήματα που ακολούθησε, τα στοιχεία εκτέλεσης του γενετικού αλγορίθμου και την προτεινόμενη λύση. Πιο κάτω παρουσιάζεται ένα απόσπασμα log αρχείου:

```
Locating Fault in progress...
Weights selected by user:
      Failed to Success:3.0
      Success to Success:1.0
      Final Slice Size:1.0
Copying source files to working directory...
Reading-sorting program slice lines...
Initial Program Slice is:
      com\mindprod\base64\Example.java Line No:13        java.lang.String originalString = arg[0];
      com\mindprod\base64\Example.java Line No:14        byte[] sendBytes = originalString.getBytes( "8859_1" );
      com\mindprod\base64\Example.java Line No:15        com.mindprod.base64.Base64 base64 = new com.mindprod.base64.Base64();
      com\mindprod\base64\Example.java Line No:17        java.lang.String encoded = base64.encode( sendBytes );
      com\mindprod\base64\Example.java Line No:18        byte[] reconstitutedBytes = base64.decode( encoded );
      com\mindprod\base64\Example.java Line No:19        java.lang.String reconstitutedString = new java.lang.String(
                                                         reconstitutedBytes, "8859_1" );
      com\mindprod\base64\Example.java Line No:22        System.out.println( "decoded:" + " " + reconstitutedString );
      com\mindprod\base64\Base64.java Line  No:12        protected final int IGNORE = -1;
      com\mindprod\base64\Base64.java Line  No:51        spec2 = '/';
      com\mindprod\base64\Base64.java Line  No:53        initTables();
      com\mindprod\base64\Base64.java Line  No:58        byte[] b = new byte[s.length() / 4 * 3];
      com\mindprod\base64\Base64.java Line  No:59        int cycle = 0;
      com\mindprod\base64\Base64.java Line  No:62        int len = s.length();
      com\mindprod\base64\Base64.java Line  No:64        for (int i = 0; i < len; i++) {
      com\mindprod\base64\Base64.java Line  No:65            int c = s.charAt( i );
      com\mindprod\base64\Base64.java Line  No:66            int value = c <= 255 ? charToValue[c] : IGNORE;
       ..
       ..
       ..
      com\mindprod\base64\Base64.java Line No:226            for (int i = 0; i < 64; i++) {
      com\mindprod\base64\Base64.java Line No:227                cv[vc[i]] = i;
      com\mindprod\base64\Base64.java Line No:231        valueToChar = vc;
      com\mindprod\base64\Base64.java Line No:232        charToValue = cv;
Initial Program Slice Size is 55
Running Genetic Algorithm ...
maxFitness= ( (NoOfSuccessTestCases * successToSuccessWeight * SuccessToSuccessScore) + (NoOfFailedTestCases *
failedToSuccessWeight * FailedToSuccessScore)) / finalSliceSizeWeight
maxFitness= ( (2 * 1.0 * 10) + (1 * 3.0 * 10)) / 1.0
maxFitness=50.0
Program slice mutators/substitutions: 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 8 * 1 * 1 * 4 * 1 * 8 * 1 * 1 * 1 * 1 * 1 * 1 * 6 * 1
* 13 * 1 * 9 * 6 * 3 * 2 * 9 * 2 * 9 * 3 * 2 * 3 * 2 * 3 * 2 * 3 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 1 * 8 * 6 * 1 * 3 * 2 * 1 * 1
Maximum GA population size for Program Slice: 195596953583616
```

```
Evolution 0 - Fittest Chromosome has fitness value 7.3076923076923075
Total Fitness Value of Current Generation is 412.3651283443636
Evolution 1 - Fittest Chromosome has fitness value 7.6
Total Fitness Value of Current Generation is 514.6641227532531
..
..
..
Evolution 108 - Fittest Chromosome has fitness value 35.0
Total Fitness Value of Current Generation is 2776.5
Evolution 109 - Fittest Chromosome has fitness value 50.0
Total Fitness Value of Current Generation is 2882.5000000000014
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
com\mindprod\base64\Base64.java Line No:147              combined |= b[i + 2] | 0xff;  - Proposed substitution:  b[i + 2] | 0xff
                                                                                                                =>   b[i + 2] & 0xff
Final Program Slice Size is 1
Fittest Chromosome has fitness value 50.0
Total execution time is:448 seconds
```

## 7.5 Αποτελέσματα

Στο υποκεφάλαιο αυτό παραθέτουμε τα αποτελέσματα από όλα τα πειράματα που έχουν διεξαχθεί με το εργαλείο. Όλα τα προγράμματα που έχουν χρησιμοποιηθεί περιέχουν εμφυτευμένα λάθη (seeded faults), τα οποία έχουμε εισαγάγει στα εν λόγω προγράμματα. Έγιναν τρεις σειρές πειραμάτων με βάση τα λάθη που έχουν εισαχθεί στα προγράμματα. Στην πρώτη σειρά πειραμάτων έχουν εισαχθεί λάθη τα οποία αντιστοιχούν σε μεταλλάξεις στο επίπεδο μεθόδου. Στη δεύτερη κατηγορία πειραμάτων έχουν εισαχθεί στα προγράμματα λάθη τα οποία αντιστοιχούν σε μεταλλάξεις στο επίπεδο κλάσης και στην τελευταία σειρά πειραμάτων έχουν εισαχθεί στα προγράμματα λάθη τα οποία δεν αντιστοιχούν σε κανένα τελεστή μετάλλαξης (βλέπε τους τελεστές μετάλλαξης που έχουν αναφερθεί στο κεφάλαιο 5.3).

Όλα τα προγράμματα που έχουν χρησιμοποιηθεί στα πειράματα είναι λύσεις σε γνωστά προγραμματιστικά προβλήματα (benchmarks).

Για όλα τα πειράματα που διεξάχθηκαν έχουν επιλεγεί τα πιο κάτω βάρη για υπολογισμό της συνάρτησης καταλληλότητας:

Failed to Success: 3.0
Success to Success: 1.0
Final Slice Size: 1.0

Τέλος πρέπει να σημειωθεί ότι όλα τα πειράματα έχουν διεξαχθεί σε μηχανή Dell Inspiron I6000 Intel Pentium M processor 1.73GHz, 2.00 GB RAM.

### 7.5.1 Σειρά Πειραμάτων Α΄

Σε αυτή την ομάδα έχουμε εμφυτέψει στα προγράμματα λάθη τα οποία αντιστοιχούν σε τελεστές μετάλλαξης σε επίπεδο μεθόδου. Έχουμε εμφυτέψει λάθη για τις ακόλουθες κατηγορίες τελεστών: (1) αριθμητικό τελεστή (arithmetic operator), (2) σχεσιακό τελεστή (relational operator), (3) τελεστή συνθήκης (conditional operator), (4) λογικό τελεστή (logical operator) και (5) ανάθεση (assignment). Στο τελευταίο πείραμα που έχει εφαρμοστεί για τελεστή ανάθεσης, το λάθος που έχει εμφυτευτεί εφαρμόζεται σε shift τελεστές, με αυτό τον τρόπο καλύψαμε και την κατηγορία των shift τελεστών μετάλλαξης. Λεπτομέρειες αναφορικά με τα πειράματα της Σειράς Ά, βλέπουμε στους Πίνακες 7.1 και 7.2.

| Αριθμός Πειράματος | Όνομα Προγράμματος | Τελεστής Μετάλλαξης | Αριθμός Γραμμών Αρχικού Προγράμματος (uncommented) | Αριθμός Γραμμών Τεμαχίου Προγράμματος Εισόδου | Αριθμός Γραμμών Τελικού Τεμαχίου | Εύρεση γραμμής λάθους; | Προτάθηκε η σωστή αντικατάσταση; |
|---|---|---|---|---|---|---|---|
| A1 | ValidateCreditCard | AODs | 183 | 16 | 1 | NAI | NAI |
| A2 | TriangleClassification | ROR | 80 | 11 | 1 | NAI | NAI |
| A3 | ValidateCreditCard | COR | 183 | 33 | 1 | NAI | NAI |
| A4 | Base64 | LOR | 293 | 55 | 1 | NAI | NAI |
| A5 | Base64 | ASR | 293 | 55 | 1 | NAI | NAI |

**Πίνακας 7.1 - Αποτελέσματα Σειράς προγραμμάτων Ά: Λάθη που αντιστοιχούν σε τελεστές μετάλλαξης σε επίπεδο μεθόδου**

| Αριθμός Πειράματος | Όνομα Προγράμματος | Μέγιστος Βαθμός Καταλληλότητας (Max Fitness) | Χώρος Αναζήτησης (Population Size) | Αριθμός Γενεών (Evolution No.) | Άθροισμα Βαθμού Καταλληλότητας (Total Fitness) | Καλύτερος Βαθμός Καταλληλότητας (Best Fitness Value) | Χρόνος εκτέλεσης (s) |
|---|---|---|---|---|---|---|---|
| A1 | ValidateCreditCard | 120.0 | 1345344000 | 200 | 6142.5 | 90.0 | 337 |
| A2 | TriangleClassification | 80.0 | 128700 | 41 | 5961.666666666666 | 80.0 | 136 |
| A3 | ValidateCreditCard | 130.0 | 243035928635899904 | 123 | 10469.999999999995 | 130.0 | 696 |
| A4 | Base64 | 50.0 | 195596953583616 | 109 | 2882.5000000000014 | 50.0 | 448 |
| A5 | Base64 | 90 | 195596953583616 | 144 | 350.91071428571433 | 90 | 191 |

**Πίνακας 7.2 - Αποτελέσματα Σειράς προγραμμάτων Ά: Τιμές για τις μεταβλητές του Γ.Α.**

**Πείραμα Α1 - ValidateCreditCard**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα ValidateCreditCard το οποίο προμηθευτήκαμε από τον ακόλουθο διαδικτυακό χώρο: http://mindprod.com/. Το εν λόγω πρόγραμμα δέχεται σαν είσοδο ένα αριθμό από x ψηφία, ο οποίος αντιστοιχεί στον αριθμό μιας πιστωτικής κάρτας και ταυτοποιεί την κάρτα, επιστρέφει δηλαδή τον προμηθευτή (vendor) της κάρτας. Το πρόγραμμα τερματίζει επιτυχώς στην περίπτωση που ταυτοποιήσει τον προμηθευτή της κάρτας με ένα από τους ακόλουθους: Visa, AMEX, Diners/Carte Blanche, JCB, MasterCard.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 57, ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον Αριθμητικό τελεστή μετάλλαξης AODs (βλέπε κεφάλαιο 5.3). Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
57  cachedLastFind = i
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
57  cachedLastFind = i++
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί δύο ανεπιτυχές και έξι επιτυχή σενάρια ελέγχου. Για τα δύο ανεπιτυχή σενάρια το

πρόγραμμα επιστρέφει λανθασμένα τον προμηθευτή “Visa”. Στις αντίστοιχες γραμμές του αρχείου δίνεται το αναμενόμενο αποτέλεσμα που είναι “AMEX” και “Discover”.

```
com\mindprod\creditcard\ValidateCreditCard.java 1
F;379999999999999;vendor_name.equals("AMEX")
S;6010222233334444;vendor_name.equals("Error: unknown credit card company")
S;4000000000000;vendor_name.equals("Visa")
S;4999999999999;vendor_name.equals("Visa")
S;4999999999998;vendor_name.equals("Visa")
S;0;vendor_name.equals("Error: not enough digits")
F;6011222233334444;vendor_name.equals("Discover")
S;9999999999999999;vendor_name.equals("Error: unknown credit card company")
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.1 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 57 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.2 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 199 - Fittest Chromosome has fitness value 90.0
Total Fitness Value of Current Generation is 6142.5
Final Program Slice is ....
com\mindprod\creditcard\ValidateCreditCard.java Line No:57
cachedLastFind = i++;  - Proposed substitution: i++ => i
Final Program Slice Size is 1
Fittest Chromosome has fitness value 90.0
Total execution time is:337 seconds
```

**Πείραμα Α2 – TriangleClassification**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί επίσης το πρόγραμμα TriangleClassification, το οποίο έχουμε προμηθευτεί από το [PA03] και είναι το γνωστό πρόγραμμα αναγνώρισης του είδους ενός τριγώνου.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 30, ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον Σχεσιακό τελεστή μετάλλαξης ROR (βλέπε κεφάλαιο 5.3). Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
30  if ((i + j <= k) || (j + k <= i) || (i + k <= j)) {
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
30  if ((i + j == k) || (j + k <= i) || (i + k <= j)) {
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί ένα ανεπιτυχές και πέντε επιτυχή σενάρια ελέγχου. Για το ανεπιτυχές σενάριο το πρόγραμμα επιστρέφει λανθασμένα την τιμή 1 για τις πλευρες 2,3,6 που αντιστοιχεί σε σκαληνό τρίγωνο ενώ το αναμενόμενο αποτέλεσμα όπως δηλώνουμε και στην αντίστοιχη γραμμή του αρχείου σεναρίων είναι 4 (δηλαδή οι συγκεκριμένες πλευρές δεν αποτελούν τρίγωνο)

```
org\testfiles\TriangClassification.java 3
F;2;3;6;result==4
S;1;2;2;result==2
```

```
S;1;1;1;result==3
S;1;2;3;result==4
S;6;9;4;result==1
S;0;0;0;result==4
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.1 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μία γραμμή κώδικα, τη γραμμή 30 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.2 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 41 - Fittest Chromosome has fitness value 80.0
Total Fitness Value of Current Generation is 5961.666666666666
Final Program Slice is ....
org\testfiles\TriangClassification.java Line No:30            if (i +
j == k || j + k <= i || i + k <= j) {  -       Proposed substitution:
i + j == k  =>   i + j <= k
Final Program Slice Size is 1
Fittest Chromosome has fitness value 80.0
Total execution time is:136 seconds
```

**Πείραμα Α3 – ValidateCreditCard**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί επίσης το πρόγραμμα ValidateCreditCard, όπως και στο πείραμα Α1. Για τους σκοπούς όμως του εν λόγω πειράματος καλέσαμε την διαδικασία του προγράμματος η οποία ελέγχει κατά πόσο ο αριθμός που δίνεται σαν είσοδος στο πρόγραμμα είναι έγκυρος αριθμός πιστωτικής κάρτας. Για να αποφασίσει κατά πόσο είναι έγκυρος ο αριθμός το πρόγραμμα κατά πρώτον ταυτοποιεί την κάρτα, επιστρέφει δηλαδή τον προμηθευτή (vendor) της κάρτας και κατά δεύτερο με βάση τον προμηθευτή αποφασίζει αν ο αριθμός που δόθηκε είναι έγκυρος

εφαρμό ζοντας τον αλγόριθμο που ισχύει για τον συγκεκριμένο προμηθευτή για υπολογισμό του ψηφίου ελέγχου (check digit).

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 56, ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης συνθήκης COR (βλέπε κεφάλαιο 5.3). Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
56  if (ranges[i].low <= creditCardNumber && creditCardNumber <= ranges[i].high)
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
56  if (ranges[i].low <= creditCardNumber || creditCardNumber <= ranges[i].high)
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί δύο ανεπιτυχή και επτά επιτυχή σενάρια ελέγχου. Για τα δύο ανεπιτυχή σενάρια το πρόγραμμα λανθασμένα επιστρέφει ότι οι συγκεκριμένοι αριθμού αποτελούν έγκυρους αριθμούς πιστωτικής κάρτας.

```
com\mindprod\creditcard\ValidateCreditCard.java 1
S;379999999999999;validCard==false
S;6010222233334444;validCard==false
S;4000000000006;validCard==true
S;378888888888858;validCard==true
S;6011222233334444;validCard==true
```

```
F;5000000005111;validCard==false
F;5000008105111;validCard==false
S;10000000009;validCard==false
S;999999999999999987;validCard==false
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.1 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μία γραμμή κώδικα, τη γραμμή 56 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.2 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 123 - Fittest Chromosome has fitness value 130.0
Total Fitness Value of Current Generation is 11019.333333333332
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
com\mindprod\creditcard\ValidateCreditCard.java Line No:56
if (ranges[i].low <= creditCardNumber || creditCardNumber <=
ranges[i].high) {  - Proposed substitution:  ranges[i].low <=
creditCardNumber || creditCardNumber <= ranges[i].high  =>
ranges[i].low <= creditCardNumber && creditCardNumber <=
ranges[i].high

Final Program Slice Size is 1
Fittest Chromosome has fitness value 130.0
Total execution time is:696 seconds
```

**Πείραμα Α4 – Base64**

**Περιγραφή Προγράμματος:**

Για τους σκοπούς του παρόντος πειράματος έχει χρησιμοποιηθεί το πρόγραμμα Base64, το οποίο έχουμε προμηθευτεί από τον διαδικτυακό χώρο: http://mindprod.com/. Το πρόγραμμα δέχεται σαν είσοδο μια σειρά από χαρακτήρες (string). Στη συνέχεια

επεξεργάζεται το string κωδικοποιώντας το σε κωδικοποίηση Base64. Τέλος αποκωδικοποιεί το string και το επιστρέφει σαν έξοδο του προγράμματος. Οπότε το πρόγραμμα αναμένεται να επιστρέψει την είσοδο του, διαφορετικά δεν έχει τερματίσει επιτυχώς.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 147, ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον Λογικό τελεστή μετάλλαξης LOR (βλέπε κεφάλαιο 5.3). Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
147  combined |= b[i + 2] & 0xff
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
147  combined |= b[i + 2] | 0xff
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί ένα ανεπιτυχές και δύο επιτυχή σενάρια ελέγχου. Για το ανεπιτυχές σενάριο το πρόγραμμα παίρνει σαν είσοδο τη σειρά χαρακτήρων `"abc"` και αναμένεται αφού κωδικοποιήσει και αποκωδικοποιήσει τους χαρακτήρες αυτούς με τον αλγόριθμο base64 να επιστρέφει τους ίδιους χαρακτήρες σαν έξοδο ενώ επιστρέφει μια άλλη σειρά χαρακτήρων.

```
com\mindprod\base64\Example.java 1
```

```
F;abc;reconstitutedString.equals("abc")
S;ab;reconstitutedString.equals("ab")
S;a;reconstitutedString.equals("a")
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.1 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μία γραμμή κώδικα, τη γραμμή 147 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.2 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 109 - Fittest Chromosome has fitness value 50.0
Total Fitness Value of Current Generation is 2882.5000000000014
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
com\mindprod\base64\Base64.java Line No:147            combined |=
b[i + 2] | 0xff;  - Proposed substitution:  b[i + 2] | 0xff  =>   b[i
+ 2] & 0xff
Final Program Slice Size is 1
Fittest Chromosome has fitness value 50.0
Total execution time is:448 seconds
```

**<u>Πείραμα Α5 – Base64</u>**

**Περιγραφή Προγράμματος:**

Για τους σκοπούς του παρόντος πειράματος έχει χρησιμοποιηθεί όπως και στο προηγούμενο πείραμα το πρόγραμμα Base64.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 144, ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή Ανάθεσης μετάλλαξης ASR (βλέπε κεφάλαιο 5.3). Ωστόσο επειδή το λάθος που έχει εμφυτευτεί εφαρμόζεται σε shift τελεστές, με αυτό τον τρόπο καλύψαμε και την κατηγορία των shift τελεστών μετάλλαξης.
Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
144  combined <<= 8
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
144  combined >>= 8
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί μόνο τρία ανεπιτυχή και κανένα επιτυχές σενάριο ελέγχου, μιας και το σφάλμα που έχει εισαχθεί μετατρέπει όλα τα σενάρια ελέγχου σε ανεπιτυχή.

```
com\mindprod\base64\Example.java 1
F;encodethissentence;reconstitutedString.equals("encodethissentence")
F;helloworld;reconstitutedString.equals("helloworld")
F;123456789;reconstitutedString.equals("123456789")
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.1 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος μόνο μία γραμμή κώδικα, τη γραμμή 144 και επιπλέον το

εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.2 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 64 - Fittest Chromosome has fitness value 90.0
Total Fitness Value of Current Generation is 2882.5000000000014
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
com\mindprod\base64\Base64.java Line No:144             combined >>=
8;  - Proposed substitution: combined >>= 8 => combined <<= 8
Final Program Slice Size is 1
Fittest Chromosome has fitness value 90
Total execution time is:191 seconds
```

### 7.5.2 Σειρά Πειραμάτων Β΄

Σε αυτή την ομάδα έχουμε εμφυτέψει στα προγράμματα λάθη τα οποία αντιστοιχούν σε τελεστές μετάλλαξης στο επίπεδο κλάσης. Έχουν εμφυτευτέψει λάθη που σχετίζονται με κληρονομικότητα και με ειδικά χαρακτηριστικά της προγραμματιστικής γλώσσας Java. Στον Πίνακας 7.3 βλέπουμε τα πειράματα που έγιναν για αυτή τη σειρά προγραμμάτων.

| Αριθμός Πειράματος | Όνομα Προγράμματος | Τελεστής Μετάλλαξης | Αριθμός Γραμμών Αρχικού Προγράμματος (uncommented) | Αριθμός Γραμμών Τεμαχίου Προγράμματος Εισόδου | Αριθμός Γραμμών Τελικού Τεμαχίου | Εύρεση γραμμής λάθους; | Προτάθηκε η σωστή αντικατάσταση; |
|---|---|---|---|---|---|---|---|
| B1 | PersonSortedList | IHD | 430 | 57 | 1 | NAI | NAI |
| B2 | Shapes | IOP | 486 | 53 | 1 | NAI | NAI |
| B3 | Graph-Shortest Path | JTI | 859 | 80 | 1 | NAI | NAI |
| B4 | Graph-Shortest Path | JSD | 859 | 79 | 1 | NAI | NAI |
| B5 | PersonSortedList | EOC | 430 | 53 | 1 | NAI | NAI |
| B6 | OrderSet | EAM | 384 | 85 | 1 | NAI | NAI |

**Πίνακας 7.3 - Αποτελέσματα Σειράς προγραμμάτων ΄Β: Λάθη που αντιστοιχούν σε τελεστές μετάλλαξης σε επίπεδο κλάσης**

| Αριθμός Πειράματος | Όνομα Προγράμματος | Μέγιστος Βαθμός Καταλληλότητας (Max Fitness) | Χώρος Αναζήτησης (Population Size) | Αριθμός Γενεών (Evolution No.) | Άθροισμα Βαθμού Καταλληλότητας (Total Fitness) | Καλύτερος Βαθμός Καταλληλότητας (Best Fitness Value) | Χρόνος εκτέλεσης (s) |
|---|---|---|---|---|---|---|---|
| B1 | PersonSortedList | 90.0 | 241920 | 28 | 5578.666666666667 | 90.0 | 81 |
| B2 | Shapes | 130.0 | 2970565811896320000 | 199 | 9643.833333333334 | 120.0 | 677 |
| B3 | Graph-Shortest Path | 90.0 | 65773397606400000 | 102 | 5300.0 | 90.0 | 392 |
| B4 | Graph-Shortest Path | 140.0 | 4110837350400000 | 180 | 10992.333333333336 | 140.0 | 778 |
| B5 | PersonSortedList | 130.0 | 32256 | 31 | 8009.166666666668 | 130.0 | 95 |
| B6 | OrderSet | 60.0 | 19200 | 29 | 1985.0 | 60.0 | 37 |

**Πίνακας 7.4 - Αποτελέσματα Σειράς προγραμμάτων Ά: Τιμές για τις μεταβλητές του Γ.Α.**

**Πείραμα B1 - PersonSortedList**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα PersonSortedList το οποίο προμηθευτήκαμε από το [WDS09]. Το εν λόγω πρόγραμμα δέχεται σαν είσοδο ένα σύνολο από αριθμούς που αντιστοιχούν στον αριθμό ταυτότητας μερικών ατόμων. Στη συνέχεια τους βάζει σε μια λίστα ταξινομώντας τους με βάση τον αριθμό ταυτότητας. Ένα αντικείμενο τύπου person μπαίνει στη λίστα μόνο αν ο αριθμός ταυτότητας του δεν περιέχεται ήδη σε αυτή. Τέλος το πρόγραμμα επιστρέφει τον αριθμό ατόμων που υπάρχουν στη λίστα. Το πρόγραμμα αυτό χρησιμοποιεί κληρονομικότητα για αναπαράσταση διαφορετικών κατηγοριών ατόμων όπως φοιτητής, εργαζόμενος κτλ.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 12 του αρχείου «Student.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης IHD (βλέπε κεφάλαιο 5.3). Συγκεκριμένα έχουμε εισάγει την μεταβλητή id στην κλάση παιδί Student η οποία με αυτό τον τρόπο κάνει hide την αντίστοιχη μεταβλητή στην κλάση πατέρα Person. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος Student.java**

```
    ....
  double gpa;
  public Student( java.lang.Integer id, java.lang.String n, int ag, java.lang.String ad,
java.lang.String p, double g ){
    ....
```

**Λανθασμένη Έκδοση Προγράμματος Student.java**

```
    ....
    double gpa;
12  public java.lang.Integer id = new java.lang.Integer( 0 );
    public Student( java.lang.Integer id, java.lang.String n, int ag, java.lang.String ad,
java.lang.String p, double g ){
    ....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί δύο ανεπιτυχή και τρία επιτυχή σενάρια ελέγχου.

```
weiss\bookcode\PersonSortedList.java 5
F;111111;232323;111222;445566;888555;listSize==5
S;111111;232323;111222;111111;111111;listSize==3
F;111111;232323;111222;111111;444666;listSize==4
S;564433;232323;111222;564433;564433;listSize==3
S;111111;0;111222;0;0;listSize==3
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 12 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση, σε αυτή την περίπτωση τη διαγραφή δηλαδή της γραμμής 12. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 28 - Fittest Chromosome has fitness value 90.0
Total Fitness Value of Current Generation is 5578.666666666667
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
weiss\bookcode\Student.java Line No:12     public java.lang.Integer id
= new java.lang.Integer( 0 );  - Proposed substitution: public
java.lang.Integer id = new java.lang.Integer( 0 ); is deleted.

Final Program Slice Size is 1
Fittest Chromosome has fitness value 90.0
Total execution time is:81 seconds
```

**Πείραμα B2 - Shapes**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα Shapes το οποίο προμηθευτήκαμε από το [WDS09]. Το εν λόγω πρόγραμμα δέχεται σαν είσοδο τέσσερις αριθμούς που αντιστοιχούν στις διαστάσεις ενός κύκλου (η πρώτη παράμετρος), ενός τετραγώνου (η δεύτερη παράμετρος) και ενός ορθογωνίου (η τρίτη και η τέταρτη παράμετρος). Το πρόγραμμα υπολογίζει το εμβαδόν του κάθε σχήματος με βάση τις τιμές εισόδου και στη συνέχεια βάζει τα αντικείμενα τύπου shape σε ένα δέντρο ταξινόμησης (SplayTree) με βάση το εμβαδόν τους. Τέλος επιστρέφει το όνομα του σχήματος με το μεγαλύτερο εμβαδόν.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 19 του αρχείου «Circle.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης IOP (βλέπε κεφάλαιο 5.3). Συγκεκριμένα έχουμε αντιστρέψει τον κώδικα των γραμμών 19 και 20 έτσι ώστε να καλείται η μέθοδος υπολογισμού του εμβαδού πριν την ανάθεση τιμών στις μεταβλητές x1 και y1 που αντιστοιχούν στις πλευρές του σχήματος. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος Circle.java**

```
....
19 super.setValues( x1, y1 );
20 Area();
....
```

**Λανθασμένη Έκδοση Προγράμματος Circle.java**

```
....
19 Area();
20 super.setValues( x1, y1 );
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί τρία ανεπιτυχή και τέσσερα επιτυχή σενάρια ελέγχου.

```
shapes\ShapesSplayTree.java 4
F;10.0;2.0;2.0;3.0;maxAreaShapeType.equals("Circle")
S;1.0;2.0;2.0;3.0;maxAreaShapeType.equals("Rectangle")
S;3.0;8.0;2.0;2.0;maxAreaShapeType.equals("Square")
F;22.0;4.0;5.0;1.0;maxAreaShapeType.equals("Circle")
S;7.0;12.0;6.0;4.0;maxAreaShapeType.equals("Square")
S;2.0;4.0;3.0;30.0;maxAreaShapeType.equals("Rectangle")
F;5.0;6.0;6.0;7.0;maxAreaShapeType.equals("Circle")
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 19 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση, μετακίνηση δηλαδή της γραμμής έτσι ώστε να καλείται η μέθοδος μετά την ανάθεση τιμών στις μεταβλητές x1 και y1. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 199 - Fittest Chromosome has fitness value 120.0
Total Fitness Value of Current Generation is 9643.833333333334
Final Program Slice is ....
shapes\Circle.java Line No:19         Area();  - Proposed
substitution: Overridden method call at first line.
Final Program Slice Size is 1
Fittest Chromosome has fitness value 120.0
Total execution time is:677 seconds
```

**Πείραμα Β3 - Graph-Shortest Path**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα Graph-Shortest Path το οποίο προμηθευτήκαμε από το [WDS09]. Το εν λόγω πρόγραμμα δέχεται σαν είσοδο ένα αρχείο στο οποίο ορίζεται ο γράφος, για κάθε ακμή του γράφου ορίζεται και μια τιμή, το κόστος της ακμής. Επιπλέον κατά το κάλεσμα του προγράμματος δίνονται σαν είσοδος τρεις χαρακτήρες, οι πρώτοι δύο αντιστοιχούν στην αρχή και το τέλος της ζητούμενης διαδρομής και ο τρίτος τον αλγόριθμο που θα χρησιμοποιηθεί για υπολογισμό του συντομότερου μονοπατιού και του κόστους που αντιστοιχεί σε αυτό. Το πρόγραμμα επιστρέφει πρώτα την ακολουθία κόμβων που αποτελεί το συντομότερο μονοπάτι και κατά δεύτερο το κόστος διαδρομής σύμφωνα με τις τιμές εισόδου.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 17 του αρχείου «Edge.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης JTI (βλέπε κεφάλαιο 5.3). Συγκεκριμένα αφαιρέσαμε τη λέξη κλειδί this κατά την ανάθεση της τοπικής μεταβλητής cost. Κατά αυτόν τον τρόπο η τιμή της παραμέτρου cost ανατίθεται στην ίδια την παράμετρο αντί στην τοπική μεταβλητή cost. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος Edge.java**

```
....
17 this.cost=cost;
....
```

**Λανθασμένη Έκδοση Προγράμματος Edge.java**

```
....
17 cost=cost;
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί δύο ανεπιτυχή και τρία επιτυχή σενάρια ελέγχου.

```
weiss\bookcode\Graph.java 4
F;C:\\graph1.txt;A;E;d;success==true&&g.lastPath.equals("A-B-E")
&&g.lastCost==23.0
S;C:\\graph1.txt;A;E;u;success==true&&g.lastPath.equals("A-B-E")
&&g.lastCost==2.0
S;C:\\graph1.txt;B;D;u;success==true&&g.lastPath.equals("B-E-D")
&&g.lastCost==2.0
S;C:\\graph1.txt;A;A;d;success==true&&g.lastPath.equals("A")&&g.lastCost==0.0
F;C:\\graph1.txt;B;C;n;success==true&&g.lastPath.equals("B-E-D-C")
&&g.lastCost==64.0
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 17 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 102 - Fittest Chromosome has fitness value 90.0
Total Fitness Value of Current Generation is 5300.0
Final Program Slice is ....
weiss\bookcode\Edge.java Line No:17         cost = cost;  - Proposed
substitution: cost  -->  this.cost
Final Program Slice Size is 1
Fittest Chromosome has fitness value 90.0
Total execution time is:392 seconds
```

## Πείραμα Β4 - Graph-Shortest Path

### Περιγραφή Προγράμματος:

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα Graph-Shortest Path όπως και στο προηγούμενο πείραμα.

### Περιγραφή Λάθους:

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 24 του αρχείου «Vertex.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης JSD (βλέπε κεφάλαιο 5.3). Συγκεκριμένα έχει εισαχθεί η λέξη κλειδί static στον ορισμό της μεταβλητής scratch, με αυτό τον τρόπο όλα τα αντικείμενα της κλάσης Vertex έχουν την ίδια τιμή για τη συγκεκριμένη μεταβλητή. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος Vertex.java**

```
....
24 public int scratch=0
....
```

**Λανθασμένη Έκδοση Προγράμματος Vertex.java**

```
....
24 public static int scratch=0
....
```

### Σενάρια Ελέγχου:

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί τρία ενεπιτυχή και πέντε επιτυχή σενάρια ελέγχου.

```
weiss\bookcode\Graph.java 4
F;C:\\graph1.txt;A;D;d;success==true&&g.lastPath.equals("A-B-E-D")
&&g.lastCost==66.0
S;C:\\graph1.txt;A;E;u;success==true&&g.lastPath.equals("A-B-E")
&&g.lastCost==2.0
S;C:\\graph1.txt;B;D;u;success==true&&g.lastPath.equals("B-E-D")
&&g.lastCost==2.0
S;C:\\graph1.txt;A;A;d;success==true&&g.lastPath.equals("A")&&g.lastCost==0.0
S;C:\\graph1.txt;B;C;n;success==true&&g.lastPath.equals("B-E-D-C")
&&g.lastCost==64.0
S;C:\\graph1.txt;B;E;d;success==true&&g.lastPath.equals("B-E")
&&g.lastCost==11.0
F;C:\\graph1.txt;A;E;d;success==true&&g.lastPath.equals("A-B-E")
&&g.lastCost==23.0
F;C:\\graph1.txt;D;E;n;success==true&&g.lastPath.equals("D-B-E")
&&g.lastCost==34.0
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 17 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 180 - Fittest Chromosome has fitness value 140.0
Total Fitness Value of Current Generation is 10992.333333333336
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
weiss\bookcode\Vertex.java Line No:24     public static int scratch =
0;  - Proposed substitution: static is deleted
Final Program Slice Size is 1
Fittest Chromosome has fitness value 140.0
Total execution time is:778 seconds
```

## Πείραμα B5 - PersonSortedList

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα PersonSortedList όπως και στο πρώτο πείραμα.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 83 του αρχείου «Person.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης EOC (βλέπε κεφάλαιο 5.3). Συγκεκριμένα έχει αντικατασταθεί η μέθοδος equals με τον τελεστή ==, με αυτό ν τον τρόπο η σύγκριση γίνεται σε επίπεδο reference αντί περιεχομένου. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος Person.java**

```
....
83  boolean equals  = otherId.equals(thisId)
....
```

**Λανθασμένη Έκδοση Προγράμματος Person.java**

```
....
83  boolean equals = otherId == thisId
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί τρία ανεπιτυχή και τέσσερα επιτυχή σενάρια ελέγχου.

```
weiss\bookcode\PersonSortedList.java 4
F;111111;232323;445566;111111;listSize==3
S;111222;232323;445566;111111;listSize==4
```

```
S;888671;222222;445566;111333;listSize==4
F;222222;222222;445566;222222;listSize==2
S;382222;909012;676722;111111;listSize==4
S;337801;100000;836477;111333;listSize==4
F;333333;333333;333333;333333;listSize==1
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 83 και επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 31 - Fittest Chromosome has fitness value 130.0
Total Fitness Value of Current Generation is 8009.166666666668
Maximum possible fitness value is reached! Interrupting GA...
Final Program Slice is ....
weiss\bookcode\Person.java Line No:83            boolean equals =
otherId == thisId;  - Proposed substitution: otherId == thisId =>
otherId.equals(thisId)

Final Program Slice Size is 1
Fittest Chromosome has fitness value 130.0
Total execution time is:95 seconds
```

**Πείραμα Β6 - OrderSet**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα OrderSet το οποίο έχουμε προμηθευτεί από τον παρακάτω διαδικτυακό χώρο: http://www.cse.unl.edu/~galileo/sir. Το πρόγραμμα δέχεται σαν είσοδο ταξινομημένα arrays και επιστρέφει το array που περιέχει τα κοινά στοιχεία των δύο arrays εισόδου.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 260 του αρχείου «OrderSet.java», ώστε να περιλαμβάνει λάθος που να αντιστοιχεί στον τελεστή μετάλλαξης EAM (βλέπε κεφάλαιο 5.3). Συγκεκριμένα έχει αντικατασταθεί η κλίση της μεθόδου getSetLast() με την κλήση της μεθόδου getActualSize(). Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος OrderSet.java**

```
....
260  int size2 = s2.getSetLast() + 1;
....
```

**Λανθασμένη Έκδοση Προγράμματος OrderSet.java**

```
....
260 int size2 = s2.getActualSize() + 1;
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί μόνο δύο ανεπιτυχή σενάρια ελέγχου, μιας και η εισαγωγή του σφάλματος δεν δίνει κανένα επιτυχές σενάριο.

```
sir\ordset\TestOrdSet.java 1
F;1,2,4,8,11,12|12,15,16,20,22;resultStr.equals("1-2-4-8-11-12-15-16-20-22")
F;1,2,3,4,5|5,6,7,8;resultStr.equals("1-2-3-4-5-6-7-8")
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.3 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει μόνο μια γραμμή κώδικα, τη γραμμή 260 και

επιπλέον το εργαλείο προτείνει την σωστή αντικατάσταση. Επιπλέον στον Πίνακας 7.4 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 29 - Fittest Chromosome has fitness value 60.0
Total Fitness Value of Current Generation is 1985.0
Final Program Slice is ....
sir\ordset\OrdSet.java Line No:260         int size2 = s2.getActualSize() + 1;
- Proposed substitution: s2.getActualSize() => s2.getSetLast()
Final Program Slice Size is 1
Fittest Chromosome has fitness value 60.0
Total execution time is:37 seconds
```

### 7.5.3 Σειρά Πειραμάτων Γ΄

Σε αυτή την ομάδα έχουμε εμφυτέψει στα προγράμματα λάθη τα οποία ΔΕΝ αντιστοιχούν σε κάποιο από τους τελεστές που χρησιμοποιεί το εργαλείο μας.

| Αριθμός Πειράματος | Όνομα Προγράμματος | Τελεστής Μετάλλαξης | Αριθμός Γραμμών Αρχικού Προγράμματος (uncommented) | Αριθμός Γραμμών Τεμαχίου Προγράμματος Εισόδου | Αριθμός Γραμμών Τελικού Τεμαχίου | Εύρεση γραμμής λάθους; | Προτάθηκε η σωστή αντικατάσταση; |
|---|---|---|---|---|---|---|---|
| Γ1 | FindMax | N/A | 45 | 14 | 3 | ΝΑΙ | ΟΧΙ |
| Γ2 | TriangleClassification | N/A | 80 | 16 | 4 | ΟΧΙ | ΟΧΙ |

**Πίνακας 7.5 - Αποτελέσματα Σειράς προγραμμάτων Γ: Λάθη που ΔΕΝ αντιστοιχούν σε τελεστές μετάλλαξης που χρησιμοποιεί το εργαλείο**

| Αριθμός Πειράματος | Όνομα Προγράμματος | Μέγιστος Βαθμός Καταλληλότητας (Max Fitness) | Χώρος Αναζήτησης (Population Size) | Αριθμός Γενεών (Evolution No.) | Άθροισμα Βαθμού Καταλληλότητας (Total Fitness) | Καλύτερος Βαθμός Καταλληλότητας (Best Fitness Value) | Χρόνος εκτέλεσης (s) |
|---|---|---|---|---|---|---|---|
| Γ1 | FindMax | 70.0 | 1672704 | 200 | 5446.499999999999 | 70.0 | 81 |
| Γ2 | TriangleClassification | 120.0 | 152980164480 | 200 | 8798.833333333332 | 120.0 | 474 |

**Πίνακας 7.6 - Αποτελέσματα Σειράς προγραμμάτων Γ: Τιμές για τις μεταβλητές του Γ.Α.**

**Πείραμα Γ1 - FindMax**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα FindMax το οποίο έχουμε προμηθευτεί από το [PA03]. Το εν λόγω πρόγραμμα δέχεται σαν είσοδο τέσσερις αριθμούς και επιστρέφει τον μεγαλύτερο από αυτούς.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 21, αντικαθιστώντας μια μεταβλητή με μια άλλη του ίδιου τύπου. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
21  max = num4;
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
21  max = num3;
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί ένα ανεπιτυχές και τέσσερα επιτυχή σενάρια ελέγχου.

```
org\testfiles\FindMax.java 4
F;1;2;3;4;returni==4
S;8;7;6;5;returni==8
```

```
S;10;8;9;5;returni==10
S;1;2;6;5;returni==6
S;3;5;4;11;returni==11
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.5 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει τρεις γραμμές κώδικα, μια εκ των οποίων και η γραμμή 21. Παρόλο που ο αλγόριθμος δεν έχει βρει την σωστή αντικατάσταση μιας και αυτή η περίπτωση δεν καλύπτεται από τις μεταλλάξεις που χρησιμοποιούμε, έχει επιτύχει να διαλέξει αυτή τη γραμμή με τη βοήθεια άλλων μεταλλάξεων. Αυτό δείχνει ότι ο αλγόριθμος ακόμα και σε αυτή την περίπτωση είναι βοηθητικός αφού με συνδυασμό άλλων τελεστών αλλάζει τη ροή του προγράμματος με αποτέλεσμα να μετατρέπει τα αποτυχημένα σενάρια σε επιτυχή και να διατηρεί τα επιτυχή σενάρια. Επιπλέον στον Πίνακας 7.6 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
Evolution 199 - Fittest Chromosome has fitness value 70.0
Total Fitness Value of Current Generation is 5425.833333333333
Final Program Slice is ....
org\testfiles\FindMax.java Line No:13         if (num1 > num2) {  -
Proposed substitution: num2 => ++num2
org\testfiles\FindMax.java Line No:19             max = num3;  -
Proposed substitution: num3 => num3++
org\testfiles\FindMax.java Line No:21                 max = num3;  -
Proposed substitution: num3 => ++num3
Final Program Slice Size is 3
Fittest Chromosome has fitness value 70.0
Total execution time is:81 seconds
```

**<u>Πείραμα Γ2 - TriangleClassification</u>**

**Περιγραφή Προγράμματος:**

Για το πείραμα αυτό έχει χρησιμοποιηθεί το πρόγραμμα TriangleClassification το οποίο έχει χρησιμοποιηθεί και στο πείραμα Α2.

**Περιγραφή Λάθους:**

Το πρόγραμμα έχει τροποποιηθεί, στην γραμμή 46, στην οποία αντικαταστάθηκε η σταθερά 2 με την τιμή 4. Παρακάτω παρατίθεται η σωστή έκδοση και η λανθασμένη έκδοση του προγράμματος.

**Αρχική Έκδοση Προγράμματος**

```
....
46  tri = 2;
....
```

**Λανθασμένη Έκδοση Προγράμματος**

```
....
46  tri = 4;
....
```

**Σενάρια Ελέγχου:**

Πιο κάτω παραθέτουμε το αρχείο το οποίο έχει δοθεί σαν είσοδος στο εργαλείο και περιέχει τα σενάρια ελέγχου. Για το συγκεκριμένο παράδειγμα έχουν δοθεί δύο ανεπιτυχή και έξι επιτυχή σενάρια ελέγχου.

```
org\testfiles\TriangClassification.java
3
F;4;3;3;result==2
F;1;2;2;result==2
S;1;1;1;result==3
S;1;2;3;result==4
S;6;9;4;result==1
S;3;1;1;result==4
S;0;0;0;result==4
```

```
S;1;2;-3;result==4
```

**Αποτελέσματα:**

Όπως βλέπουμε και στον Πίνακας 7.5 αλλά και στο log αρχείο που παράγεται με την εκτέλεση του αλγορίθμου (πιο κάτω παρατίθεται ένα μέρος αυτού του αρχείου) το τελικό τεμάχιο προγράμματος περιέχει τέσσερις γραμμές κώδικα. Η γραμμή 46 όμως δεν συμπεριλαμβάνεται σε αυτές. Είναι λοιπόν αυτή μια περίπτωση που ο αλγόριθμος δεν κατάφερε με τις μεταλλάξεις που έχει στη διάθεση του να αλλάξει τη ροή του προγράμματος. Επιπλέον στον Πίνακας 7.6 παρουσιάζονται οι τιμές για διάφορες μεταβλητές του γενετικού αλγορίθμου.

```
....
Evolution 199 - Fittest Chromosome has fitness value 110.0
Total Fitness Value of Current Generation is 8798.833333333332
Final Program Slice is ....

org\testfiles\TriangClassification.java Line No:29         if (tri == 0) {  -
Proposed substitution: tri => tri--

org\testfiles\TriangClassification.java Line No:36             if (tri > 3) {
- Proposed substitution: tri => tri--

org\testfiles\TriangClassification.java Line No:39                 if (tri ==
1 && i + j > k) {  - Proposed substitution:  tri == 1  =>   tri != 1

org\testfiles\TriangClassification.java Line No:42                     if (tri
== 2 && i + k > j) {  - Proposed substitution:  tri == 2 && i + k > j  =>
!(tri == 2 && i + k > j)

Final Program Slice Size is 4
Fittest Chromosome has fitness value 110.0
Total execution time is:474 seconds
```

Στο κεφάλαιο που ακολουθεί γίνεται μια αξιολόγηση των πιο πάνω αποτελεσμάτων αλλά και την μεθοδολογίας γενικότερα. Επίσης εντοπίζονται κάποιες αδυναμίες που προτείνονται σαν μελλοντική εργασία.

# Κεφάλαιο 8

# Αξιολόγηση Εργασίας

---



---

## 8.1 Αξιολόγηση παρούσας εργασίας : Συνεισφορά και Σύγκριση με άλλες εργασίες

Στην εργασία αυτή παρουσιάσαμε μία μεθοδολογία για εντοπισμό σφάλματος, η οποία παίρνοντας σαν είσοδο το τεμάχιο προγράμματος που δημιουργεί ο δυναμικός τεμαχισμός προσπαθεί να μειώσει το μέγεθος αυτού του τεμαχίου και να δώσει στον προγραμματιστή όσο το δυνατό λιγότερες γραμμές κώδικα, μέσα στις οποίες να περιέχεται και η λανθασμένη δήλωση. Επιπλέον στόχος της μεθοδολογίας είναι η εύρεση της πιθανής αντικατάστασης που διορθώνει το σφάλμα.

Στο προηγούμενο κεφάλαιο έχουμε δει τρεις σειρές πειραμάτων που έχουν διεξαχθεί, η κατηγοριοποίηση των πειραμάτων έχει γίνει με βάση το είδος του σφάλματος. Κρίναμε λοιπόν ορθό να αξιολογήσουμε την μεθοδολογία αναφερόμενοι σε κάθε σειρά πειραμάτων ξεχωριστά, δηλαδή σε κάθε τύπο σφάλματος.

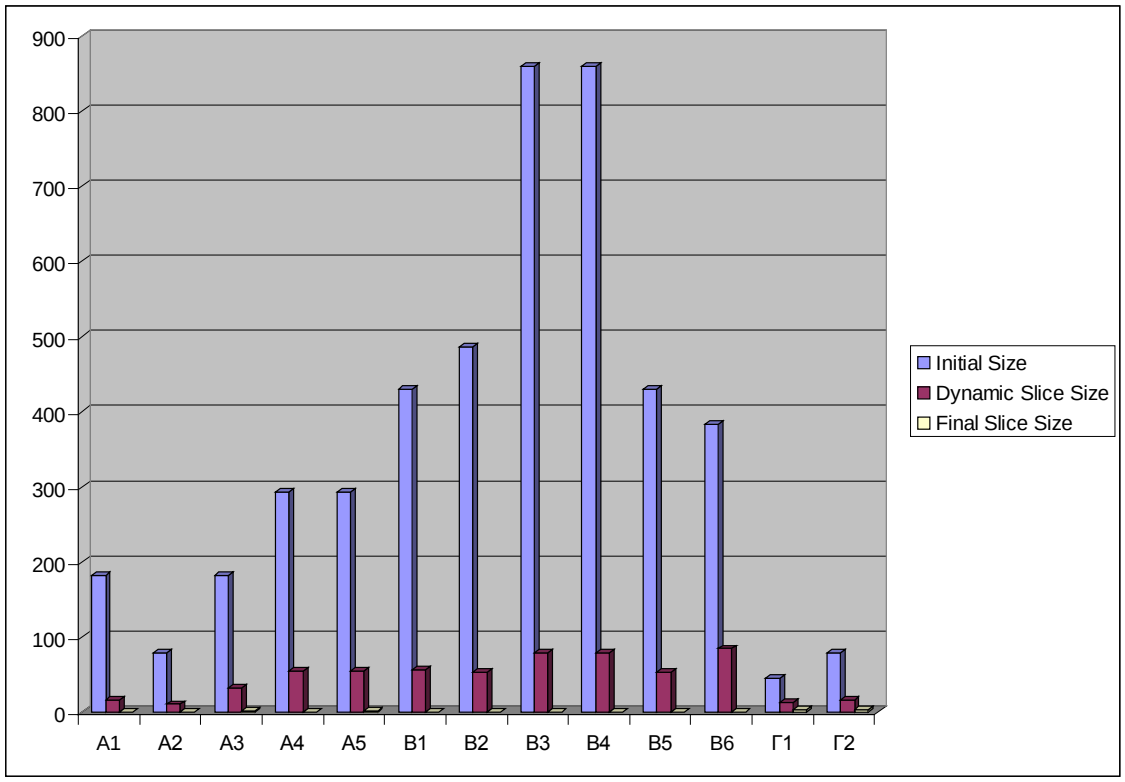
Στην σειρά πειραμάτων Α΄, έχουν χρησιμοποιηθεί προγράμματα στα οποία έχουμε εισαγάγει σφάλματα που αντιστοιχούν σε τελεστές μετάλλαξης σε επίπεδο μεθόδου. Είναι δηλαδή κοινά προγραμματιστικά λάθη που συναντώνται σε όλες τις γλώσσες προγραμματισμού. Τα αποτελέσματα των πειραμάτων που έχουν διεξαχθεί έδειξαν ότι το εργαλείο εντοπίζει το σφάλμα σε όλες τις περιπτώσεις προτείνοντας μάλιστα και τη σωστή αντικατάσταση για τη διόρθωση του σφάλματος. Επιπλέον, το μέγεθος του τελικού τεμαχίου ξεπέρασε τις αρχικές μας προσδοκίες μιας αφού περιέχει μόνο τη γραμμή του σφάλματος.

Τα αποτελέσματα της σειράς πειραμάτων Β΄, είναι επίσης ενθαρρυντικά μιας και για όλες τις περιπτώσεις το εργαλείο εντοπίζει το σφάλμα, προτείνοντας τη σωστή αντικατάσταση και το τελικό τεμάχιο που δίνεται στον προγραμματιστή περιέχει μόνο τη γραμμή του σφάλματος. Για τη συγκεκριμένη σειρά πειραμάτων έχουν χρησιμοποιηθεί προγράμματα στα οποία έχουν εισαχθεί σφάλματα που αντιστοιχούν σε τελεστές μετάλλαξης σε επίπεδο κλάσης. Στην ουσία μιλούμε για κοινά προγραμματιστικά λάθη που συναντώνται σε αντικειμενοστραφή γλώσσες προγραμματισμού, μάλιστα ένα υποσύνολο αυτών είναι σφάλματα που συναντώνται ειδικά σε Java προγράμματα.

Στην τελευταία σειρά πειραμάτων έχουν εξετασθεί προγράμματα στα οποία έχουμε εισαγάγει σφάλματα που δεν αντιστοιχούν σε συγκεκριμένους τελεστές, από αυτούς που υποστηρίζονται από το εργαλείο. Παρατηρούμαι ότι σε αυτή τη κατηγορία το εργαλείο μας παρόλο που έχει να εντοπίσει σφάλματα για τα οποία δεν είναι 'προετοιμασμένο' υπάρχει μεγάλη πιθανότητα να εντοπίσει τη γραμμή που περιέχει το σφάλμα. Η δυνατότητα του εργαλείου έγκειται στο γεγονός ότι οι τελεστές που εφαρμόζονται στη γραμμή του λάθους μπορούν να αλλάξουν τη ροή του προγράμματος οδηγώντας το σε επιτυχία. Άρα αν και δεν έχουμε τη σωστή αντικατάσταση, δίνεται στον προγραμματιστή η γραμμή που περιέχει το σφάλμα. Είναι αξιοσημείωτο επίσης ότι και σε αυτές τις περιπτώσεις το μέγεθος του τελικού τεμαχίου προγράμματος αν και μπορεί να είναι μεγαλύτερο από ότι στις προηγούμενες δύο κατηγορίες είναι αρκετά μικρό ώστε να μπορούμε να πούμε ότι το εργαλείο μειώνει σημαντικά το μέγεθος του τεμαχίου που δίνει στον προγραμματιστή. Είναι φυσικά αναμενόμενο για αυτή την κατηγορία σφαλμάτων ότι θα υπάρξουν και περιπτώσεις για τις οποίες η προτεινόμενη μεθοδολογία δεν θα μπορέσει να εντοπίσει το σφάλμα, όπως και στο πείραμα Γ2 που αφορά αντικατάσταση σταθεράς.

Για την αξιολόγηση της προτεινόμενης μεθοδολογία εκτός από την ικανότητα της στο να εντοπίζει τα λάθη άλλη σημαντική παράμετρος είναι το μέγεθος του τελικού τεμαχίου προγράμματος. Στην περίπτωση μας θα συγκρίνουμε το μέγεθος του τελικού τεμαχίου τόσο με το μέγεθος του αρχικού προγράμματος, όσο και με το μέγεθος του τεμαχίου προγράμματος που δημιουργεί ο δυναμικός αλγόριθμος τεμαχισμού. Στο πιο κάτω

σχεδιάγραμμα (Διάγραμμα 8.1) βλέπουμε μια γραφική αναπαράσταση αυτών των μεγεθών. Είναι φανερό ότι η μεθοδολογία μας μειώνει δραστικά το μέγεθος του τελικού τεμαχίου.




**Διάγραμμα 8.4: Αναπαράσταση μεγέθους Αρχικού προγράμματος, Δυναμικού τεμαχίου και Τελικού τεμαχίου**

Το τελικό τεμάχιο κυμαίνεται στο **0.1 – 9 %** του μεγέθους του αρχικού προγράμματος, ενώ το αντίστοιχο ποσοστό μεταξύ του τελικού τεμαχίου και του δυναμικού τεμαχίου προγράμματος κυμαίνεται στο διάστημα **1 – 30%**. Μια πρώτη παρατήρηση όσον αφορά την αποτελεσματικότητα της προτεινόμενης μεθοδολογίας, που προκύπτει από τη σύγκριση των πιο πάνω μεγεθών, είναι ότι ικανοποιεί την ελάχιστη απαίτηση από πλευράς αποτελεσματικότητας αφού είναι πολύ πιο αποτελεσματική από ότι η χρήση Δυναμικού τεμαχισμού από μόνη της. Σε επόμενο στάδιο θα συγκρίνουμε το μέγεθος του τεμαχίου προγράμματος που δίνεται στον προγραμματιστή από το εργαλείο μας με το μέγεθος του τεμαχίου που δίνεται χρησιμοποιώντας άλλα εργαλεία για εντοπισμό σφάλματος.

Είναι σημαντικό να σημειώσουμε ότι όσο πιο πλήρης είναι τα σενάρια ελέγχου όσον αφορά την κάλυψη διαφορετικών μονοπατιών του προγράμματος, τόσο πιο επιτυχή είναι και τα αποτελέσματα της προτεινόμενης μεθοδολογίας, με την έννοια ότι έχουμε μικρότερο μέγεθος στο τελικό μας τεμάχιο. Η χρήση λοιπόν εργαλείων εύρεσης σεναρίων ελέγχου για τα προγράμματα θα ήταν βοηθητική. Ωστόσο, ο προγραμματιστής μπορεί και χωρίς τη χρήση κάποιου εργαλείου παραγωγής σεναρίων ελέγχου, να δώσει αρχικά στο πρόγραμμα τα σενάρια που έχει παρατηρήσει κατά τις εκτελέσεις του προγράμματος, επιτυχή και ανεπιτυχή. Στη συνέχεια μετά την πρώτη εκτέλεση του αλγορίθμου για εντοπισμό του σφάλματος, σε περίπτωση που το μέγεθος του τελικού τεμαχίου είναι σχετικά μεγάλο, αφού παρατηρήσει τις γραμμές που περιέχονται στη τελικού τεμαχίου, μπορεί να προσθέσει επιπλέον σενάρια και να επανεκτελέσει τον αλγόριθμο.

Η παρούσα μεθοδολογία εκτός από τα σφάλματα τα οποία έχουν σαν αποτέλεσμα την αλλοίωση του αναμενόμενου αποτελέσματος, εντοπίζει επίσης τις περιπτώσεις που το πρόγραμμα δεν τερματίζει ή δεν φτάνει στη γραμμή που μας ενδιαφέρει λόγο οποιουδήποτε λάθους κατά την εκτέλεση (runtime error). Επιπρόσθετα η δυνατότητα του εργαλείου δυναμικού προγραμματισμού JSlice το οποίο χρησιμοποιείται για την δημιουργία του δυναμικού τεμαχίου προγράμματος, να εντοπίζει τα execution omission errors (λάθη παράληψης) υλοποιώντας ένα επαυξημένο αλγόριθμο δυναμικού τεμαχισμού [WR04], επεκτείνει το σύνολο των σφαλμάτων που εντοπίζονται από την μεθοδολογία μας συμπεριλαμβάνοντας και τα execution omission errors.

Πρέπει να σημειωθεί επίσης ότι υπάρχουν περιπτώσεις για τις οποίες το σφάλμα που υπάρχει στο πρόγραμμα είναι τέτοιο που να μην έχουμε κανένα επιτυχές σενάριο. Παρόλο που ο προγραμματιστής δίνει μόνο ένα αριθμό ανεπιτυχών σεναρίων ο αλγόριθμος εντοπίζει το σφάλμα και η μεθοδολογία μας αποδεικνύεται εξίσου αποδοτική και σε αυτή την περίπτωση (βλέπε πείραμα B6).

Είναι επίσης σημαντικό να αναφέρουμε ότι το εργαλείο που έχει αναπτυχθεί υποστηρίζει προγράμματα Java που αποτελούνται από πολλά αρχεία, ενταγμένα σε πακέτα. Επιπλέον χάριν της χρήσης της δήλωσης *assert* για αποτίμηση του αναμενόμενου

αποτελέσματος, ο προγραμματιστής μπορεί να εκφράσει το αναμενόμενο αποτέλεσμα όχι απλά με χρήση απλών αριθμητικών ή λεκτικών τελεστών σύγκρισης όπως: `x==3` ή `a=="abc"`, αλλά με χρήση οποιασδήποτε έγκυρης έκφρασης σύμφωνα με τα συμφραζόμενα του κώδικα π.χ. *`x.getMax(a,b,c)==a && y.getAverage(d,e)=3.`*

Το εργαλείο παρέχει στον χρήστη την ευελιξία να διαλέξει τους τελεστές μετάλλαξης που θα χρησιμοποιηθούν για τον εντοπισμό του λάθους. Στην περίπτωση που κατά την αρχική εκτέλεση του αλγορίθμου ο προγραμματιστής παρατηρήσει ότι στο τεμάχιο προγράμματος περιέχονται πολλές γραμμές λόγω εφαρμογής συγκεκριμένων τελεστών μετάλλαξης, μπορεί να επιλέξει στην επόμενη εκτέλεση του αλγορίθμου να μην συμπεριλάβει τον τελεστή αυτό. Για παράδειγμα αν στη γραμμή του κριτηρίου μας το λανθασμένο αποτέλεσμα ήταν `x = 3` ενώ το αναμενόμενο αποτέλεσμα ήταν `x = 4` η εφαρμογή του τελεστή μετάλλαξης $AOI_U$ μπορεί να δώσει το αναμενόμενο αποτέλεσμα, χωρίς στην ουσία να εντοπίζει το λάθος. Άλλη περίπτωση για την οποία ο προγραμματιστής θα πρέπει να επιλέξει την μη εφαρμογή του τελεστή $AOI_U$ είναι όταν εφαρμόζεται σε ένα επαναληπτικό κόμβο δημιουργώντας ατέρμονο βρόγχο (infinite loop). Ωστόσο η περίπτωση αυτή θα μπορούσε να αντιμετωπιστεί προγραμματιστικά από το εργαλείο μας, όπως θα δούμε και στο επόμενο υποκεφάλαιο για τη μελλοντική εργασία.

Στην βιβλιογραφία δεν έχει βρεθεί καμία άλλη δουλειά που να συνδυάζει τον δυναμικό τεμαχισμό, τον έλεγχο με χρήση μετάλλαξης (mutation testing) και τους γενετικούς αλγορίθμους για εντοπισμό σφάλματος σε java προγράμματα.

Ωστόσο έχουν γίνει αρκετές προσπάθειες για ανάπτυξη εργαλείων για εντοπισμό σφαλμάτων σε java προγράμματα, όπως το Bandera [CDHLPRZ00], ESC/Java [FLLNSS02], FindBugs [HP03], JLint [JLINT10] και PMD [PMDJ10]. Στο [RAF04] γίνεται μια αναφορά σε διάφορα τέτοια εργαλεία και προτείνεται η χρήση ενός meta-tool το οποίο μπορεί να συνδυάζει τα αποτελέσματα που δίνονται από τα διάφορα εργαλεία, μιας και το κάθε εργαλείο επικεντρώνεται στον εντοπισμό διαφορετικών ειδών λαθών. Τα εργαλεία αυτά αναλύουν τον κώδικα με βάση κάποιους συντακτικούς κανόνες ή με βάση την ροή δεδομένων του προγράμματος και παράγουν διάφορα μηνύματα

(warnings) για πιθανά λάθη στον κώδικα. Στην πράξη όμως δεν είναι και τόσο βοηθητικά αφού ο αριθμός των μηνυμάτων που παράγουν είναι εξαιρετικά μεγάλος και ο προγραμματιστής πρέπει να ελέγξει ένα-ένα τα μηνύματα αυτά. Επίσης παρατηρήθηκε ότι για το ίδιο παράδειγμα κώδικα μερικά εργαλεία μπορεί να μην δώσουν κανένα μήνυμα ενώ άλλα να δώσουν ένα μεγάλο αριθμό μηνυμάτων. Επιπλέον στα εργαλεία αυτά παρατηρούνται false positives , δηλαδή μηνύματα που υποδεικνύουν λάθος σε ορθό κώδικα καθώς και false negatives, δηλαδή δεν παράγουν μηνύματα για λανθασμένο κώδικα.

Στο [HS04] παρουσιάζεται ένα βελτιστοποιημένος αλγόριθμος για τεμαχισμό Java προγραμμάτων. Ο αλγόριθμος αυτός διαχειρίζεται περιπτώσεις όπου αντικείμενα περνιούνται σαν παράμετροι σε μεθόδους. Ωστόσο η εν λόγο εργασία σκοπό έχει απλώς την κατασκευή καλύτερων φετών προγράμματος χωρίς να ασχολείται με τον εντοπισμό σφάλματος.

Στη συνέχεια παραθέτουμε τις πιο αξιόλογες εργασίες που έχουμε εντοπίσει στη βιβλιογραφία και ασχολούνται με τον εντοπισμό σφάλματος. Αυτές οι εργασίες αποτελούν υποσύνολο των εργασιών που έχουν αναφερθεί στο Κεφάλαιο 1 και έχουν επιλεχθεί για αντιπαραβολή με την παρούσα εργασία.  μιας και για αυτές υπάρχει διαθέσιμη η πληροφορία για τις μετρικές αξιολόγησης. Λαμβάνοντας υπόψη το γεγονός ότι τα πειράματα που έχουν διεξαχθεί για αυτές τις εργασίες αναφέρονται σε διαφορετικά προγράμματα από ότι τα πειράματα για την προτεινόμενη μεθοδολογία, μάλιστα οι πλείστες από τις εν λόγω εργασίες δεν αναφέρονται σε java προγράμματα, έχουμε αποφασίσει να  αντιπαραβάλουμε κάθε τέτοια εργασία με την δική μας ως προς την ικανότητα της να εντοπίζει λάθη (όταν υπάρχει αυτή η πληροφορία) και ως προς το μέγεθος του παραγόμενου τεμαχίου προγράμματος σε σχέση με το αρχικό πρόγραμμα.

Τα πειράματα που έχουν γίνει στα πλαίσια της εργασίας «*Locating Faulty Code Using Failure - Inducing Chops*» [GHZG05] έδειξαν ότι τα failure inducing chops που παράγονται έχουν μέγεθος ίσο με το 7 με 14% του μεγέθους ολόκληρου του προγράμματος. Επιπλέον κατά μέσο όρο η λανθασμένη δήλωση περιέχεται στο failure inducing chop στο 4 3 με 1 0 0 %των περιπτώσεων. Συγκρίνοντας λοιπόν αυτή την

εργασία με τη δική μας βλέπουμε ότι η παρούσα εργασία υπερτερεί μιας και το μέγεθος του τελικού τεμαχίου κυμαίνεται στο 0.1 - 9% του μεγέθους του αρχικού προγράμματος και επιπλέον το λάθος περιέχεται σε όλα τα τεμάχια προγράμματος πλην μιας. Τέλος η παρούσα εργασία προτείνει στον χρήστη ποια αντικατάσταση να εφαρμόσει για την επίλυση του σφάλματος, πράγμα που δεν γίνεται από την εργασία των Gupta , Zhang, He και Gupta.

Στην εργασία *«The Efficiency of Critical Slicing in Fault Localization»* [KWRK05], το μέγεθος των critical slices κυμαίνεται από 2.7 μέχρι 50% του μεγέθους του αρχικού προγράμματος. Ωστόσο μόνο το 59.33% των critical slices περιέχει το σφάλμα. Παρόλο που τα προγράμματα που έχουν χρησιμοποιηθεί για την παρούσα εργασία είναι διαφορετικά από τα μικρά σε μέγεθος Fortran προγράμματα που χρησιμοποιήθηκαν στην [KWRK05], είναι φανερό ότι η παρούσα εργασία υπερτερεί παράγοντας μικρότερα τεμάχια προγράμματος, για τις οποίες η πιθανότητα να περιέχουν το λάθος είναι μεγαλύτερη και επιπρόσθετα προτείνεται και η αντικατάσταση για τη διόρθωση του σφάλματος.

Για τη εργασία των Zhang και Gupta, με τίτλο: *«Locating Faults through Automated Predicate Switching»* [ZGG06b] η πειραματική εργασία έχει δείξει ότι το μέγεθος του τεμαχίου προγράμματος του bidirectional chop (BiChop) το οποίο αποτελεί το τεμάχιο προγράμματος που πρέπει να ελέγξει ο προγραμματιστής για να εντοπίσει το σφάλμα, κυμαίνεται μεταξύ 0.24 μέχρι 53.34% του αρχικού προγράμματος. Ωστόσο οι δηλώσεις του BiChop δίνονται στον προγραμματιστή ταξινομημένες με σειρά προτεραιότητας έτσι ώστε να εξετάσει πρώτα τις δηλώσεις που έχουν περισσότερη πιθανότητα να περιέχουν το λάθος. Στα πειράματα που διεξήχθησαν ο εντοπισμός του σφάλματος επιτεύχθηκε μετά την εξέταση 1 μέχρι 3 δηλώσεων του BiChop. Συγκρίνοντας με την παρούσα εργασία στην οποία τις περισσότερες φορές επιστρέφεται 1 δήλωση προς εξέταση και στη χειρότερη 4 μπορούμε να πούμε ότι οι δύο εργασίες έχουν κοντινά αποτελέσματα, με τη διαφορά ότι η παρούσα εργασία εκτός από τον εντοπισμό του λάθους, προτείνει και την αντικατάσταση που διορθώνει το σφάλμα.

Στο [ZGG07] *«Locating Faulty Code By Multiple Points Slicing»* το μέγεθος των multiple points dynamic slices κυμαίνεται από 0.02 μέχρι 6.98% του μεγέθους του αρχικού προγράμματος. Παρόλο που τα ποσοστά μεγέθους των dynamic slices που παράγονται στην εργασία αυτή είναι κάπως καλύτερα από τα αντίστοιχα ποσοστά της δικής μας εργασίας, στην παρούσα εργασία το τελικό τεμάχιο περιέχει τις πλείστες φορές μόνο τη γραμμή του λάθος και επιπλέον προτείνει στον χρήστη την αντικατάσταση για την επίλυση του σφάλματος .

Στην εργασία των Zhang και Gupta, «A Study of Effectiveness of Dynamic Slicing in Locating Real Faults» [ZGG07b], το μέγεθος των φετών προγράμματος που πρέπει να εξετάσει ο προγραμματιστής με σκοπό τον εντοπισμό του σφάλματος κυμαίνεται από 0.07 μέχρι 8.52% του μεγέθους του αρχικού προγράμματος. Όπως είναι φανερό είναι μικρότερο σε σχέση με το μέγεθος των φετών προγράμματος που παράγονται είτε με Data Slicing (0.45-37.78%) είτε με Full Slicing (0.90-63.18%). Σε σύγκριση με την παρούσα εργασία τα ποσοστά μεγέθους είναι πολύ κοντινά, ωστόσο στη δική μας εργασία η πιθανότητα να περιέχεται το λάθος στο τελικό τεμάχιο είναι ψηλότερη αφού με βάση τα αποτελέσματα η πιθανότητα αυτή είναι 92 3 % σε αντίθεση με την άλλη δουλείας που η πιθανότητα αυτή κυμαίνεται στο διάστημα 0.45-63.18%. Επιπλέον η παρούσα εργασία προτείνει την διόρθωση του λάθους.

Ανακεφαλαιώνοντας, η αντιπαραβολή με τις πιο πάνω εργασίες έχει δείξει ότι η παρούσα εργασία αποτελεί μια πολλά υποσχόμενη μεθοδολογία, μιας και μειώνει το μέγεθος του τεμαχίου προγράμματος σε τέτοιο βαθμό, ώστε ο προγραμματιστής στις περισσότερες περιπτώσεις να έχει στη διάθεση του μόνο την δήλωση που περιέχει το σφάλμα. Επιπλέον η παρούσα εργασία κάνει την υπέρβαση σε σχέση με τις εργασίες που έχουμε αναφέρει μιας και προτείνει την αντικατάσταση που διορθώνει το σφάλμα, σημειώνεται ότι καμία από τις εργασίες που έχουν σκοπό τον εντοπισμό σφάλματος δεν καταπιάνεται με την διόρθωση του εν λόγο σφάλματος.

## 8.2 Περιορισμοί και Μελλοντική Εργασία

Όπως κάθε άλλη προτεινόμενη μεθο **δ**λογία έτσι και η δική μας έχει κάποιους περιορισμούς τους οποίους πρέπει να λάβουμε υπόψη για μελλοντική εργασία.

Είναι φανερό από τα αποτελέσματα που έχουν παρουσιαστεί στο Κεφάλαιο 6, ότι η μεθοδολογία όσον αφορά τις περιπτώσεις των σφαλμάτων που αντιστοιχούν σε τελεστές αντικατάστασης που υλοποιούνται από το εργαλείο, εντοπίζει πάντα το σφάλμα επιστρέφοντας στο τελικό τεμάχια πολύ λίγες γραμμές κώδικα (μια ή δύο γραμμές). Επιπλέον προτείνει τη σωστή αντικατάσταση για την διόρθωση του σφάλματος. Για τις υπόλοιπες περιπτώσεις σφαλμάτων, αν και έχει καλά αποτελέσματα, είναι αναμενόμενο ότι δεν μπορεί να προτείνει την σωστή αντικατάσταση για διόρθωση του σφάλματος. Θα μπορούσαμε λοιπόν να επεκτείνουμε το σύνολο των τελεστών που υλοποιούνται από το εργαλείο συμπεριλαμβάνοντας τελεστές που αντιστοιχούν σε περισσότερες περιπτώσεις αντικαταστάσεων, για παράδειγμα αντικατάσταση σταθερών τιμών (constant replacement) ή αντικατάσταση μιας μεταβλητής σε μία μέθοδο με μία άλλη μεταβλητή του ιδίου τύπου που υπάρχει στη μέθοδο αυτή.

Μια άλλη παράμετρος για την οποία υπάρχουν σημαντικά περιθώρια βελτίωσης είναι η επίδοση (performance) του αλγορίθμου. Για τον σκοπό αυτό θα μπορούσαμε να υλοποιήσουμε τον γενετικό αλγόριθμο και ειδικά την συνάρτηση καταλληλότητας χρησιμοποιώντας multithreading για παράλληλη επεξεργασία.

Επιπλέον με την εν λόγω μετατροπή θα μπορούσε να αντιμετωπιστεί εύκολα και ένας άλλος περιορισμός της υλοποίησης μας. Συγκεκριμένα, υπάρχο ιν περιπτώσεις για τις οποίες η εφαρμογή ενός τελεστή έχει ως αποτέλεσμα την δημιουργία ατέρμονου βρόγχου στο υπό εξέταση πρόγραμμα. Για παράδειγμα αν εφαρμοστεί η αντικατάσταση της μεταβλητής `i`, με `i--`, όπως φαίνεται στο πιο κάτω παράδειγμα προγράμματος.

**Αρχικός κώδικας**

```
while i<10 {
      …
      i = i + 1;
}
```

**Μετάλλαξη κώδικα**

```
while i<10 {
      …
      i = i-- + 1;
}
```

Στην παρούσα φάση το ενδεχόμενο αυτό δεν αντιμετωπίζεται από την υλοποίηση, ο χρήστης μπορεί φυσικά να διακόψει την εκτέλεση του αλγορίθμου και να επιλέξει την μη-εφαρμογή του τελεστή στην επόμενη εκτέλεση. Με την υλοποίηση της συνάρτησης καταλληλότητας ως ξεχωριστής διεργασίας (thread) θα μπορούσαμε σε τέτοια περίπτωση δημιουργίας ατέρμονου βρόγχου να σταματούμε τη διεργασία αν δεν τερματίσει μέσα σε συγκεκριμένο χρονικό διάστημα.

Αναφορικά με τα παραδείγματα προγραμμάτων τα οποία έχουν χρησιμοποιηθεί ως πειραματική εργασία για την αξιολόγηση της μεθοδολογίας μας, στα πλαίσια της παρούσας εργασίας έχουν χρησιμοποιηθεί προγράμματα με ‘εμφυτευμένα’ (seeded) λάθη. Σε ένα επόμενο στάδιο θα μπορούσαν να χρησιμοποιηθούν προγράμματα με πραγματικά λάθη (real faults).

Όπως έχει αναφερθεί και πιο πάνω η επιλογή των κατάλληλων σεναρίων ελέγχου που δίνονται σαν είσοδος στον αλγόριθμο μας είναι άμεσα συνδεδεμένη με το μέγεθος του τελικού τεμαχίου προγράμματος. Η επιλογή σεναρίων ελέγχου για να έχουμε υψηλό ποσοστό κάλυψης των μονοπατιών ενός προγράμματος αποτελεί ερευνητικό θέμα από μόνη της. Η χρήση εργαλείων εύρεσης σεναρίων ελέγχου για τα προγράμματα θα ήταν βοηθητική. Για τον σκοπό αυτό θα μπορούσαν επίσης να χρησιμοποιηθούν τα εργαλεία ελέγχου με χρήση μετάλλαξης όπως το muJava για αξιολόγηση της επάρκειας των σεναρίων ελέγχου, όπως είναι άλλωστε και ο σκοπός δημιουργίας τους. Υπάρχουν επίσης και εργαλεία για αυτόματη παραγωγή σεναρίων ελέγχου όπως το Kiasan και το Korat [BKM02]. Σημειώνεται όμως ότι ο προγραμματιστής θα πρέπει να

κατηγοριοποιήσει τα σενάρια αυτά σε επιτυχή και ανεπιτυχή. Επιπρόσθετα μεγάλη θα μπορούσε να είναι η συνεισφορά εργαλείων όπως το Java Coverage Analyzer [JCA10] που μετρά τον βαθμό κάλυψης μονοπατιών ενός προγράμματος από συγκεκριμένα σενάρια ελέγχου.

Τέλος, έχοντας ως ερευνητικό ερέθισμα την εύρεση σεναρίων ελέγχου, θα μπορούσαμε επίσης να χρησιμοποιήσουμε το εργαλείο μας για την αξιολόγηση των σεναρίων ελέγχου που υπάρχουν για ένα πρόγραμμα. Εισάγοντας διάφορα σφάλματα στο πρόγραμμα μπορούμε να ελέγχουμε κατά πόσο με χρήση των υπαρχόντων σεναρίων το εργαλείο μας μπορεί να εντοπίσει το σφάλμα δίνοντας το μικρότερο δυνατό τεμάχιο προγράμματος.

## Βιβλιογραφία


[ADS93] H. Agrawal, R. A. DeMillo, and E. H. Spafford, “Debugging with dynamic slicing and backtracking”, Software – Practice and Experience, vol. 23, no. 6, pp. 589-616, 1993.

[AH90] H. Agrawal and J.R. Horgan, “Dynamic program slicing”, Proceedings of ACM SIGPLAN Conference on programming Language Design and Implementation, White Plains, New York, U.S.A., ACM Press, pp. 246-256, 1990.

[AHKL93] Hirabal Agrawal, Joseph R. Horgan, Edward W. Krauser and Saul A. London, “Incremental regression testing ”, Proceedings of the Conference on Software Maintenance, pp.1-10, 1993.

[AHLW95] Hiralal Agrawal, Joseph R. Horgan, Saul London, W. Eric Wong, “Fault Localization Using Execution Slices and Dataflow Tests”, Software Reliability Engineering, Proceedings Sixth International Symposium, Volume-Issue: 24-27, pp: 143 – 151, October 1995.

[ALSP09] Apache Logging Services Project – Apache log4j, http://logging.apache.org/log4j/, Accessed: December 2009.

[BCHW05] Sue Black, Steve Counsell, Tracy Hall, Paul Wernick, “Using Program Slicing to identify Faults in Software”, Dagstuhl, Seminar Nº 05451, November 2005.

[BDGHKK05] Dave Binkleya, Sebastian Danicicb, Tibor Gyimσthyc, Mark Harmand, Akos Kissc, Bogdan Korele, “Theoretical foundations of dynamic program slicing”, February 2005.

[BE93] J. Beck and D. Eichmann, “Program and interface slicing for reverse engineering”, Proceedings of 15th International Conference on Software Engineering, Baltimore, Maryland, USA, IEEE CS Press, pp. 509-518, 1993.

[BHRS00] D Binkley, M Harman, L.R. Raszewski, and C. Smith, “An empirical study of amorphous slicing as a program comprehension support tool”, Proceedings of 8th International Workshop on Program Comprehension, Limerick, Ireland, IEEE CS Press, pp. 161-170, 2000.

[BI98] D. Binkley, “The application of program slicing to regression testing”, Information and Software Technology, Vol. 40, No. 11\12, pp.583-594, 1998.

[BKM02] C. Boyapati, S. Khurshid, and D. Marinov, “Korat: Automated Testing Based on Java Predicates,” Proc. Int’l Symp. Software Testing and Analysis, pp. 123-133, July 2002.

[CCL98] Gerardo Canfora, Aniello Cimitile, Andrea De Lucia, “Conditioned program slicing”, Information and Software Technology Special Issue on Program Slicing, Volume 40, p. 595-607. Elsevier Science B.V., 1998.

[CCLL94] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, “Software salvaging based on conditions”, Proceedings of International Conference on Software Maintenance, Victoria, British Columbia, Canada, IEEE CS Press, pp. 424-433, 1994.

[CDHLPRZ00] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Pasareanu, Robby, and H. Zheng, “Bandera: Extracting Finite-state Models from Java Source Code”, In Proceedings of the 22nd International Conference on Software Engineering, pages 439–448, Limerick Ireland, June 2000.

[CLLF94] G. Canfora, A. De Lucia, G.A. Di Lucca, and A. R. Fasolino, “Slicing large programs to isolate reusable functions”, Proceedings of EUROMICRO Conference, Liverpool, U.K., IEEE CS Press, pp. 140-147, 1994.

[CLM95] A. Cimitile, A. De Lucia, and M. Munro, “Identifying reusable functions using specification driven program slicing: a case study”, Proceedings of International Conference on Software Maintenance, Opio (Nice), France, IEEE CS Press, pp. 124-133, 1995.

[CLM96] A. Cimitile, A. De Lucia, M. Munro, “A specification driven slicing process for identifying reusable functions”, Journal of Software Maintenance: Research and Practice, vol. 8, n. 3, pp. 145-178, 1996.

[CMN91] J.-D. Choi, B.P. Miller, and R.H.B. Netzer. “Techniques for debugging parallel programs with flowback analysis”. ACM Transactions on Programming Languages and Systems, 13(4):491–530, 1991.

[CW89] J. S. Collofello and S. N. Woodfield, “Evaluationg the Effectiveness of Reliability-Assurance Techniques”, Journal of Systems and Software 9(3): 191-195, 1989.

[DFHH01] S. Danicic, C. Fox, M. Harman and R. Hierons, “Backward Conditioning: a new program specialization technique and its application to program comprehension”, Proceedings of 9th International Workshop on Program Comprehension, Toronto, Canada, IEEE CS Press, pp. 89-97, 2001.

[DKN01] Y. Deng, S. Kothari, and Y. Namara, “Program slicing browser”, Proceedings of 9th International Workshop on Program Comprehension, Toronto, Ontario, Canada, IEEE CS Press, pp. 50-59, 2001.

[DLS78] Richard A. DeMillo and Richard J. Lipton and Frederick Gerald Sayward, “Hints on Test Data Selection: Help for the Practicing Programmer”, IEEE Computer 11(4), p. 34-41, April 1978.

[DS86] R. A. DeMillo and E. H. Spafford, “The Mothra software testing environment”, presented at The 11th NASA Software Engineering Laboratory Workshop, Goddard Space Center, 1986.

[FFN91] W.B. Frakes, C. J. Fox and B. A. Nejmeh, “Software Engineering in the UNIX/C Environment”, Englewood Cliffs, New Jersey, USA: Prentice-Hall, 1991.

[FLLNSS02] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata, “Extended Static Checking for Java”, In Proceedings of the 2002 ACM SIGPLAN Conference on Programming Language Design and Implementation, pages 234–245, Berlin, Germany, June 2002.

[FRT95] John Field, G. Ramalingam, Frank Tip, “Parametric program slicing”, Proceedings of the ACM Symposium on Principles of Programming Languages, New York, pp. 379-392, 1995.

[GCLL00] G. Canfora, A. Cimitile, A. De Lucia, and G.A. Di Lucca, “Decomposing legacy programs: a First step towards migrating to client-server platforms”, The Journal of Systems and Software, vol. 54, pp. 99-110, 2000.

[GHS95] R. Gupta, M.J. Harrold and M.L. Soffa, “An approach to regression testing using slicing”, Proceedings of the Conference on Software Maintenance, Orlando, FL, USA, IEEE CS Press, pp.299-308, 1995.

[GHZG05] N. Gupta, H. He, X. Zhang, and R. Gupta, “Locating Faulty Code Using Failure-Inducing Chops”, IEEE/ACM International Conf. On Automated Software Engineering, Long Beach, CA, Nov. 2005.

[GL91] K. B. Gallagher and J. R. Lyle, “Using program slicing in software maintenance”, IEEE Transactions on Software Engineering, Vol. 17, No. 8, pp. 751-761, 1991.

[GM02] Diganta Goswami and Rajib Mall, “An efficient method for computing dynamic program slices”, Information Proc-essing Letters, 81(2), pp. 111-117, January 2002.

[GO04] L. Gallagher and A. J. Offutt, “Integration Testing of Object-oriented Components Using FSMS: Theory and Experimental Details”, Technical report ISE-TR-04-04, Department of Information and Software Engineering, George Mason University, Fairfax, VA, July 2004.

[GO89] D. E. Goldberg, “Genetic Algorithms in Search, Optimization, and Machine Learning”, Addison-Wesley, 1989.

[GO91] R. Gopal, “Dynamic program slicing based on dependence relations”, Proceedings of Conference on Software Maintenance, Sorrento, Italy, IEEE CS Press, pp. 191-200, 1991.

[GOH92] Robert Geist and A. Jefferson Offutt and Frederick C. Harris, “Estimation and Enhancement of Real-Time Software Reliability through Mutation Analysis”, IEEE Transactions on Computers, 41(5), May 1992.

[HA77] Richard Graham Hamlet, “Testing Programs with the Aid of a Compiler”, IEEE Transactions on Software Engineering, 3(4), p. 279-290, July 1977.

[HA95] Robert J. Hall, “Automatic extraction of executable program subsets by simultaneous dynamic program slicing”, Automated Software Engineering, 2(1), pp. 33-53, March 1995.

[HD95] Mark Harman and Sebastian Danicic, “Using Program Slicing to Simplify Testing”, Software Testing, Verification and Reliability, 5(3) 143-162, 1995.

[HD97] Mark Harman, Sebastian Danicic, “Amorphous program slicing”, Proceedings of IEEE International Workshop on Program Comprehension (IWPC’97), Dearborn, Michigan, pp. 70-79, May 1997.

[HDS95] M. Harman, S. Danicic, and Y. Sivagurunathan, “Program comprehension assisted by slicing and transformation”, Proceedings of 1st UK Program Comprehension Workshop, Durham, UK, 1995.

[HH01] Mark Harman, Robert M. Hierons, “An Overview of Program Slicing”, Software Focus Vol. 2, No. 3, p. 85-92, 2001.

[HHD99] R. Hierons, M. Harman, and S. Danicic, “Using program slicing to assist in the detection of equivalent mutants”, Journal of Software Testing, Verification and Reliability, vol. 9, no. 4, pp. 233-262, 1999.

[HO75] J. H. Holland, “Adaptation in Natural and Artificial Systems”, University of Michigan Press, 1975.

[HP03] D. Hovemeyer and W. Pugh, “Finding Bugs Is Easy”, http://www.cs.umd.edu/?pugh/java/bugs/docs/findbugsPaper.pdf, Accessed: January 2010.

[HPR89] S. Horwitz, J. Prins, and T. Reps, “Integrating noninterfering versions of programs”, ACM Transactions on Programming Languages and Systems, vol. 11, no. 3, pp. 345-387, 1989.

[HR94] M. J. Harrold and Gregg Rothermel. “Performing data flow testing on classes”. In Symposium on Foundations of Software Engineering, pages 154-163, New Orleans, LA, December 1994.

[HS04] C. Hammer and G. Snelting, “An improved slicer for Java”, Proc. 2004 ACM SIGPLAN-SIGSOFT Ws. Program Anal. for Software Tools and Eng. (ACM Press), 2004.

[HS91] M.J. Harrold and M.L. Soffa, “Selecting dataflow integration testing”, IEEE Software, vol 8, no.2, pp. 58-65, 1991.

[HU89] W. S. Humphrey, “Managing the Software Process”, Software Engineering Institute, Addison-Wesley, Reading, MA, 1989.

[JCA10] Java Coverage Analyser, http://www.cse.iitk.ac.in/users/jalote/download/javacoverage/index.html, Accessed: January 2010.

[JG07] Dennis Jeffrey and Neelam Gupta, “Improving Fault Detection Capability by Selectively Retaining Test Cases during Test Suite Reduction”, Software Engineering, IEEE Transactions on, Volume: 33, Issue: 2, Pages: 108-123, February 2007.

[JG08] JGAP: Java Genetic Algorithms Package, http://jgap.sourceforge.net, Accessed: March 2008

[JGHL08] Tao Jiang, Nicolas Gold, Mark Harman and Zheng Li , “Locating Dependence Structures Using Search Based Slicing”, Journal of

Information and Software Technology, Volume 50, No.12, p. 1189-1209, November 2008.

[JJF07] Sun Jirong, Li Zhshu, Ni Jiancheng, Yin Feng, “Software Fault Localization Based on Testing Requirements and Program Slice”, International Conference on Networking, Architecture, and Storage (NAS 2007), pages:168 – 176, July 2007.

[JLINT10] JLint, http://artho.com/jlint, Accessed: January 2010.

[JS08] JSlice - a Java Dynamic Slicing Tool, http://jslice.sourceforge.net/, Accessed: March 2008.

[KA98] M. Kamkar, “Application of program slicing in algorithm debugging”, Information and Software Technology, vol. 40, no. 11/12, pp. 637-646, 1998.

[KB98] B.K. Kang and J.M. Bieman, “Using design abstractions to visualize, quantify, and restructure software”, Journal of Systems and Software, vol. 42, no. 2, pp. 175-187, 1998.

[KFS93] M. Kamkar, P. Fritzson, and N. Shahmerhi, “Interpocedural dynamic slicing applied to interprocedural data flow testing”, Proceedings of Conference on Software Maintenance, Montreal, Quebec, Canada, IEEE CS Press, pp. 386-395, 1993.

[KK94] H.S. Kim and Y.R. Kwon, “Restructuring programs through program slicing”, International Journal of Software Engineering and Knowledge Engineering, vol. 4, no. 3, pp. 349-368, 1994.

[KL90] B. Korel, J. Laski, “Dynamic slicing of computer programs”, The Journal of Systems and Software, Vol. 13, no. 3, pp. 187-195, 1990.

[KO93] B. Korel, “Identifying faulty modifications in software maintenance”, Proceedings of 1st International Workshop on Automated and Algorithmic Debugging, Linköping, Sweden, Springer Verlag, New York, pp. 341-356, 1993.

[KR97] B. Korel and J. Rilling, “Dynamic program slicing in understanding of program execution”, Proceedings of 5th International Workshop on Program Comprehension, Dearborn, MI, USA, pp. 80-90, 1997.

[KR98] B. Korel and J. Rilling, “CASE and dynamic program slicing in software maintenance”, International Journal of Computer Science and Information Management, June 1998.

[KR98b] Bodgan Korel, Jurgen Rilling, “Dynamic program slicing methods”, Information and Software Technology 40 647-659, 1998.

[KS98] J. Krinke and G. Snelting, “Validation of measurement software as an application of slicing and constraint solving”, Information and Software Technology, vol. 40, no. 11/12, pp. 661-676, 1998.

[KWRK05] Z. A. Al-Khanjari, M. R. Woodward, Haider Ali Ramadhan, N. S. Kutti, “The Efficiency of Critical Slicing in Fault Localization”, Software Quality Control archive, Volume 13 , Issue 2, Pages: 129 – 153, June 2005.

[LD98] A. Lakhotia and J.C. Deprez, “Restructuring programs by tucking statements into functions”, Information and Software Technology, vol. 40, no. 11/12, pp. 677-691, 1998.

[LFM96] A. De Lucia, A.R. Fasolino, and M. Munro, “Understanding function behaviors through program slicing”, Proceedings of 4th Workshop on

Program Comprehension, Berlin, Germany, IEEE CS Press, pp. 9-18, 1996.

[LH98] D. Liang and M. J. Harrold, “Slicing objects using system dependence graphs”, In ICSM, pages 358-367, 1998.

[LV97] F. Lanubile and G. Visaggio, “Extracting reusable functions by flow graph-based program slicing”, IEEE Transactions on Software Engineering, vol. 23, no. 4, pp. 246-259, 1997.

[LW87] J.R. Lyle and M. Weiser, “Automatic program bug location by program slicing”, Proceedings of 2nd International Conference on Computers and Applications, Peking, China, pp. 877-882, 1987.

[LZ75] B. H. Liskov and S. N. Zilles, “Specification Techniques for Data Abstractions”, Software Engineering, Vol. SE-1, No. 1, p. 7-19, March 1975.

[MKO02] Y. S. Ma, Y. R. Kwon, and J. Offutt, “Inter-class mutation operators for Java”, IEEE Computer Society Press, editor, 13th International Symposium on Software Reliability Engineering, pages 352-363, November 2002.

[MMS03] G. B. Mund, Rajib Mall and Sudeshna Sarkar, “Computation of intraprocedural dynamic program slices”, Information and Software Technology, 45(8), pp. 499-512, June 2003.

[MO00] I. Moore, “Jester – a JUnit Test Tester”, Extreme Programming and Flexible Processes in Software Engineering - XP2000, 2000.

[MO05] Yu-Seung Ma, Jeff Offutt, “Description of Class Mutation Mutation Operators for Java”, November 2005.

[MO05b] Yu-Seung Ma, Jeff Offutt, “Description of Method-level Mutation Operators for Java”, November 2005.

[MTR09] “Mutation Testing Repository”, http://www.dcs.kcl.ac.uk/pg/jiayue/repository/ Accessed: December 2009.

[MY79] G. J. Myers, “The Art of Software Testing”, Wiley-Interscience, 1979.

[NEK93] J.Q. Ning, A. Engberts, and W. Kozaczynski, “Recovering reusable components from legacy systems by program segmentation”, Proceedings of 1st Working Conference on Reverse Engineering, Baltimore, Maryland, U.S.A., IEEE CS Press, pp. 64-72, 1993.

[OLRUZ96] A. J. Offutt, A. Lee, G. Rothermel, R. Untch, and C. Zapf, “An experimental determination of sufficient mutation operators”. ACM Transactions on Software Engineering Methodology, 5(2):99-118, April 1996.

[OMK05] J. Offutt, Y.-S. Ma, and Y. R. Kwon, “MuJava : An Automated Class Mutation System”, Software Testing, Verification and Reliability, vol. 15, pp. 97-133, 2005.

[OU01] A. Jefferson Offutt and Roland H. Untch “Mutation 2000: Uniting the Orthogonal Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)San Jose, California, 6-7 October 2001.

[PA03] Paul Ammann and Jeff Offutt, “Introduction to Software Testing”, February 2003.

[PAR72] David L Parnas, “On The Criteria Used In Decomposing Systems Into Modules”, Commun. Ass. Comput. Mach., Vol. 15, p. 1053-1058, December 1972.

[PMDJ10] PMD/Java, http://pmd.sourceforge.net, Accessed: January 2010.

[RAF04] Rutar, N. Almazan, C.B. Foster, J.S., “A comparison of bug finding tools for Java”, Software Reliability Engineering, 15th International Symposium, page(s): 245- 256, November 2004.

[RR03] M. Renieris and S. P. Reiss, “Fault Localization With Nearest Neighbor Queries”, International Conference on Automated Software Engineering, pp. 30-39, 2003.

[SA08] Anastasis A. Sofokleous, Andreas S. Andreou, ”Automatic, evolutionary test data generation for dynamic software testing” , Journal of Systems and Software 81(11): 1883-1898 (2008).

[SAK09] Anastasis A. Sofokleous, Andreas S. Andreou, Antonis Kourras, “Symbolic Execution for Dynamic, Evolutionary Test Data Generation”, ICEIS (1) 2009: 144-150.

[SH99] Yeong-Tae Song, Dung T. Huynh, “Forward dynamic object-oriented program slicing”, Application-Specific Systems and Software Engineering and Technology (ASSET ‘99), IEEE CS Press, pp. 230-237, 1999.

[ST06] Ron Patton, “Software Testing”, Second Edition, Sams Publishing, 2006.

[STA76] James F. Stay, “HIPO and Integrated Program Design”, IBM Systems Journal, Vol. 15, No. 2, p. 143-154, 1976.

[TIP95] Frank Tip, “A Survey of Program Slicing Techniques”, Journal of Programming Languages Vol. 3, No. 3, p. 121-189, September 1995.

[UOH93] R. Untch, A. J. Offutt, and M. J. Harrold, “Mutation analysis using program schemata”, In Proceedings of the 1993 International Symposium on Software Testing and Analysis, pages 139-148, Cambridge MA, June 1993.

[VE91] G. A. Venkatesh, “The semantic approach to program slicing”, Proceedings of the ACM Transactions on Programming Languages and Systems, 13(2), pp. 181-210, 1991.

[WDS09] “Book Samples: Mark Allen Weiss. Data Structures and Algorithm Analysis in Java”, Addison Wesley, http://users.cis.fiu.edu/~weiss/, Accessed: December 2009.

[WEI82] Mark Weiser, “Programmers Use Slices when Debugging”, Communications of the ACM, Vol. 25, No. 7, p. 446-452, July 1982.

[WEI84] Mark Weiser, “Program Slicing”, IEEE Transactions on Software Engineering, Vol. 10, p. 352-357, July 1984.

[WR04] Tao Wang and Abhik Roychoudhury, “Using Compressed Bytecode Traces for Slicing Java Programs”, ACM/IEEE International Conference on Software Engineering (ICSE), 2004.

[XCY02] Baowen Xu, Zhenqiang Chen and Hongji Yang, “Dynamic slicing object-oriented programs for debugging”, Second IEEE International Workshop on Source Code Analysis and Ma-nipulation (SCAM'02), Montreal, Canada, p. 115, Oct. 2002.

[XZWC05] Baowen Xu Ju Qian, Xiaofang Zhang, Zhongqiang Wu, Li Chen , “A Brief Survey Of Program Slicing”, ACM SIGSOFT Software Engineering Notes, Volume 30 Number2, March 2005.

[ZGG06] X. Zhang, N. Gupta and R. Gupta, “A Study of Effectiveness of Dynamic Slicing in Locating Real Faults”, Empirical Software Engineering Journal, August 2006.

[ZGG06b] Xiangyu Zhang, Neelam Gupta, Rajiv Gupta, “Locating Faults Through Automated Predicate Switching”, International Conference on Software Engineering archive Proceeding of the 28th international conference on Software engineering, Shanghai, China, Pp: 272 - 281, 2006.

[ZGG06c] Xiangyu Zhang, Neelam Gupta, Rajiv Gupta, “Pruning dynamic slices with confidence”, Conference on Programming Language Design and Implementation archive, Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation table of contents, Ottawa, Ontario, Canada, SESSION: Medley table of contents, pp:169 - 180, 2006.

[ZGG07] Xiangyu Zhang, Neelam Gupta, Rajiv Gupta, “Locating Faulty Code By Multiple Points Slicing”, Software - Practice & Experience (SPE), Volume 37, Issue 9, July 2007.

[ZGG07b] Xiangyu Zhang, Neelam Gupta, Rajiv Gupta, “A Study of Effectiveness of Dynamic Slicing in Locating Real Faults”, Empirical Software Engineering archive Volume 12 , Issue 2, pp: 143 – 160, , April 2007.

[ZGZ03] Xiangyu Zhang, Rajiv Gupta and Youtao Zhang, “Precise dynamic slicing algorithms”, 25th International Conference on Software Engineering, pp. 319-329, May 2003.

[ZH98] Jianjun Zhao, “Dynamic slicing of object-oriented programs”, Technical Report SE-98-119, Information Processing Society of Japan (IPSJ), pp. 17-23, May 1998.

[ZHGG05] X. Zhang, H. He, N. Gupta, and R. Gupta, “Experimental Evaluation of Using Dynamic Slices for Fault Location,” Sixth International Symposium on Automated and Analysis-Driven Debugging, Monterey, California, September 2005.

[ZTGG07] Xiangyu Zhang, Siraman Tallan, Neelam Gupta, Rajiv Gupta, “Towards Locating Execution Omission Errors”, ACM SIGPLAN Conference on Programming Language Design and Implementation, p: 415-424, June 2007.