ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)
στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Χρήση μεθόδου μεταλλάξεων σε προγράμματα υλοποιημένα σε Java»

Μαστοραντωνάκης Μιχαήλ

M3010016

ΑΘΗΝΑ, ΔΕΚΕΜΒΡΙΟΣ 2002

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)
στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

«Χρήση μεθόδου μεταλλάξεων σε προγράμματα υλοποιημένα σε Java»

Μαστοραντωνάκης Μιχαήλ

M3010016

Επιβλέπων: **Επίκουρος Καθηγητής Νικόλαος Μαλεύρης**

Εξωτερικός Κριτής: **Αναπληρωτής Καθηγητής Εμμανουήλ Γιακουμάκης**

ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΑΘΗΝΑ, ΔΕΚΕΜΒΡΙΟΣ 2002

Executive Summary

Testing is a very crucial phase of the software development cycle. It uses various methods targeting at the reliability improvement of the tested program through the marking of existed bugs, which is its main goal by Dijkstra. Mutation testing came out from the fault-seeding approach. One basic goal of it, like all other testing methods, is to create and evaluate test data by counting their efficiency on finding faults. Injection of known faults in the program was leading to a great number of wrong versions, something that made the method computationally expensive. The faults were being infecting the source code or the used data.

Corresponding to that is the logic of creating mutants by changing the source code only and never the data that are used for tests. This came out as a result because of two important admissions, but the number of faulty versions created is still great enough. Goal of the present study is to have mutation testing applied in programs written in JAVA language in such a way that generates the less possible mutants, while trying to accelerate anyhow the process.

The method

In more details now the method of using mutants is a powerful technique that injects faults in program units of the under test software. It helps the oracle user to create and check the efficiency of test data iteratively and interactively [OC94]. The tester defines the type and number of mutation operators used, knowing which are the most critical for the tested system [Mar89]. The technique is dealing with faults in the source code level related with the use of a language's operator, an operand or a control element [Mar89]. This results to a lot of different versions from the original program, the mutants, generated from mutant operators, which are strongly language-depended. If the output of a mutant is different from the corresponding of the original program for the same input is considered killed. In any other case the mutant has to be checked if equivalent to the original program or should be tested with new data till it outputs differently. In practise it is too difficult to find data able to kill all the possible mutations [Mar89]. The efficiency of the test data in killing mutants is measured by dividing the number of killed with the number of non-equivalent mutants and this is called Mutation Adequacy Score (MS) [UOH93].

During the past research there were various techniques focused on applying the method and evaluating its operators. Each time the goal was to reduce the number of faulty versions and the acceleration of the whole process, in such a way that would set it practically acceptable in order to be really used as part of the development cycle. Very important problem that comes out from all the studies is stated as the equivalent mutant detection. It is something that most of the approaches, apart from the Mothra system for Fortran, haven't done a lot with. Other noticeable efforts were dealing with the weak [How82], the firm [WH88], which stands between the previous one and the strong approach, the one that used schemas [UOH93] and the selective one [VM98]. Lately a lot of interest has been born for adjusting the method to the special attributes of the object-oriented languages. System integration, interface [DMM01, GM01, VMBD01] or class mutation [KCM00, KCM00b]seems to get a lot of attention from the researchers.

The proposed new approach

Through reading on the existing references a lot of advantages were spotted together with some problems that had to be eliminated as more as possible. The starting idea was to create one and only program containing all the mutants to be tested, in a way similar to the one the Mutant Schema Generator (MSG) system introduced [UOH93]. In order to make this happen, the code should be enriched with calls to statically predefined methods or dynamically created ones. At the same time calls to special methods should be added, while parsing and compiling the code, in order to make available at run-time information of that phase.

Comparing outputs from mutants related to those of the original program seamed to be a very time demanding step of the process. So an automated mechanism was looked for in order to decide for the killed or equivalent versions. Comparative checks had to be added somewhere in the program's logic to make that happen. According to Howden's theory [How82] in weak mutation is acceptable to execute all the mutants for one place when executing its code. Based on this idea the schemas were expanded with checks for the local results and were made responsible to execute all the selected mutants for their position each time they were reached.

The weak mutation though according to some studies is giving overestimated data efficiency numbers. In order to reduce this inefficiency relatively to the strong

mutation measurements the related research has been taken in mind [OL94, OLR+96]. So in the case of iterative statements we applied firm mutation [WH88], always into schema structures and by providing the right mechanism for comparing the variables state after N circles. For this check the transformation of the loop into a whole class with methods is very helpful.

Also there were not applied mutation operators that change the control flow of the program since the weak approach was followed. These operators are not suggested to be included in a system from the selective mutation either [ORZ93, VM98, MB99]. Some other operators were not included because of differences of the JAVA language with Fortran, Ada or others and because the admission about the programmer's capability was extended. A testing system is computationally very expensive which makes it applicable to safety critical programs mostly. This means that the programmers that developed these programs are capable more than the advantage and well experienced. That makes some mistakes almost impossible to occur.

Finally the proposed approach is taking advantage of the control flow of the program when creating mutants. Injection of schemas into the program doesn't mean that they have been created. This is because indirectly a path coverage analysis is done driven by the test data and there is no generation of mutants for dead or unreachable code. If the data is unable to reach a path this is no problem for this mutation system.

Conclusions

Previous theory was saying that the upper limit of the generated mutants for a program is related to the square of its variable references. For the proposed system it is also difficult to find the exact number of expected mutants. Though it is estimated that the maximum expected number is given from the below function:

$$\sum_{t=1}^{T}(VR_t\cdot(V_t-1))+\sum_{l=0}^{L}\left\lfloor\sum_{t=1}^{T}(IV_t\cdot V_t)+3\cdot IC\right\rfloor_l+8\cdot L+\sum_{s=0}^{S}(2\cdot C_s+1)+$$
$$3\cdot CR+4\cdot AOR+2\cdot LOR+2\cdot BOR+5\cdot ROR$$

Symbol	Description
VR_t	References to variables of type t
V_t	Variables of type t or casting appropriate type of t
T	Number of different variable types in program
IV_t	Number of initializations with variable use of type t.
IC	Number of initializations with constant values.
L	Number of iterative statements (while, for, do – while)
S	Number of switch-case statements
C_s	Number of case statements in the switch-case **s**
CR	Number of references to constants.
AOR	Number of uses of arithmetical operators
LOR	Number of uses of logical operators
BOR	Number of uses of binary operators
ROR	Number of uses of relational operators

Future Studies

As far as the future extensions of the present approach are concerned the special attributes of the object-oriented JAVA should be examined in order to add more language specific mutation operators in the system. It would be helpful to examine the adjustment of analytical techniques in the schemas to help in solving the equivalent detection problem. In previous work done the Constraint-Based Testing has been introduced [OP97]. This is something that might help towards a solution of the same problem.

Furthermore the use of machine learning could help in defining the most proper mutation operators associated to the critical level of the system under test. This may suppress the statically defined schemas and lead into a new transformer strategy. Also program-slicing techniques [OC94, Wei84, HHD99] as stated to previous research, if adopted from the suggested system, could help by simplifying programs resulting to a probable reduction of the mutants' number.

Περιεχόμενα

Σχήματα

Πίνακες

Περίληψη

Ο έλεγχος είναι σημαντικό κομμάτι στον κύκλο ανάπτυξης λογισμικού, που χρησιμοποιεί ποικίλες μεθόδους στην προσπάθεια για βελτίωση της αξιοπιστίας του μέσα από την ανάδειξη λαθών κάτι που σύμφωνα με τον Dijkstra είναι και ο στόχος του. Πριν ακόμα προταθεί η μέθοδος των μεταλλάξεων εφαρμοζόταν μια αντίστοιχη τεχνική αυτή της «σποράς λαθών» (fault seeding). Ένας βασικός σκοπός της, όπως και για όλες τις μεθόδους ελέγχου λογισμικού, ήταν η δημιουργία και ταξινόμηση των δεδομένων ελέγχου με κριτήριο την αποτελεσματικότητα εύρεσης λαθών. Προσθέτοντας γνωστά λάθη στο πρόγραμμα, οδηγούσε σε μεγάλο πλήθος λανθασμένων εκδόσεων κάτι που την καθιστούσε όμως υπολογιστικά ασύμφορη. Η εμφύτευση των λαθών γινόταν είτε στον κώδικα είτε στα δεδομένα με τα οποία δοκίμαζαν το πρόγραμμα.

Αντίστοιχη είναι η λογική της δημιουργίας μεταλλάξεων με την μόνη διαφορά ότι ασχολείται μόνο με τον κώδικα και όχι με τα δεδομένα. Σκοπός της εργασίας είναι να εφαρμοστεί η μέθοδος των μεταλλάξεων σε προγράμματα γραμμένα σε JAVA με τρόπο τέτοιο που να προκύπτουν όσο το δυνατό λιγότερες μεταλλάξεις και να επιταχυνθεί με όποιο τρόπο η διαδικασία.

Η μέθοδος

Στην προσπάθεια μείωσης του πλήθους των περιπτώσεων προς έλεγχο, που έδινε η μέθοδος της εμφύτευσης λαθών, έγιναν κάποιες σημαντικές παραδοχές που οδήγησαν στην μέθοδο των μεταλλάξεων. Από την μια η υπόθεση ότι ο προγραμματιστής είναι αρκετά ικανός, ώστε το πρόγραμμα του να απέχει λίγα μόνο λάθη από το επιθυμητό («σωστό»), και από την άλλη η διαπίστωση ότι δεδομένα που αποκαλύπτουν απλά λάθη έχουν την ικανότητα να ανακαλύψουν και πιο σύνθετα. Παρόλα αυτά όμως το πλήθος των περιπτώσεων, που πρέπει να ελεγχθούν, παραμένει αρκετά σημαντικό και αναζητούνται τρόποι να μειωθεί περαιτέρω.

Αναλυτικότερα η μέθοδος των μεταλλάξεων είναι μια ισχυρή τεχνική που «φυτεύει» λάθη στο υπό εξέταση λογισμικό σε επίπεδο μονάδας προγράμματος. Βοηθάει τον ελεγκτή-χρήστη να δημιουργήσει και να ελέγξει **δεδομένα ελέγχου (test data)** με επαναληπτικό (iteratively) και αλληλεπιδραστικό τρόπο προς ενίσχυση της αποτελεσματικότητας τους [OC94]. Ο υπεύθυνος του ελέγχου καθορίζει το είδος και

το πλήθος των τελεστών μετάλλαξης που θα χρησιμοποιηθούν, γνωρίζοντας ποια είναι πιο κρίσιμα για το κάθε σύστημα [Mar89]. Η τεχνική αφορά λάθη στον κώδικα, που μπορεί να υπάρχουν στην εφαρμογή κάποιου τελεστή (operator), τελεσταίου (operand) ή στοιχείο ελέγχου (control element) [Mar89]. Έτσι δημιουργούνται πολλές διαφοροποιημένες εκδόσεις του αρχικού προγράμματος, «**μεταλλάξεις**» **(mutants)**, από τους **τελεστές μετάλλαξης (mutation operators)**, οι οποίοι είναι αυστηρά εξαρτώμενοι από την γλώσσα προγραμματισμού (language-dependent). Αν τα αποτελέσματα από την εκτέλεση της μετάλλαξης είναι διαφορετικά από αυτό του πρωτότυπου προγράμματος τότε θεωρείται «**σκοτωμένη**» **(killed)**. Διαφορετικά μπορεί να αποτελεί μια «**ισοδύναμη**» **(equivalent)** έκδοση ή να απαιτεί τη δημιουργία νέων δεδομένων που θα μπορούν να το «σκοτώσουν». Στην πράξη όμως είναι πολύ δύσκολο να δημιουργηθούν δεδομένα που να ανακαλύπτουν όλες τις δυνατές μεταλλάξεις [Mar89]. Η αποτελεσματικότητα των δεδομένων στην εξόντωση μεταλλάξεων μετράται από το λόγο των μη ισοδύναμων προς το συνολικό πλήθος. Σύμφωνα με ένα δεύτερο συντελεστή που αναφέρεται στην αρθρογραφία [UOH93] ο **βαθμός καταλληλότητας (Mutation Adequacy Score - MS)** δίνεται από το λόγο των «σκοτωμένων» προς το πλήθος των μη ισοδύναμων μεταλλάξεων.

Στην πορεία των μελετών υπήρξαν διάφορες τεχνικές με σκοπό την εφαρμογή της μεθόδου και την αξιολόγηση των τελεστών μετάλλαξης. Στόχος κάθε φορά ήταν η μείωση του πλήθους των λανθασμένων εκδόσεων και η επιτάχυνση της μεθόδου, ώστε να μπορέσει να αποτελέσει και πρακτικά μέρος του κύκλου ανάπτυξης ενός λογισμικού. Σημαντικό πρόβλημα που προκύπτει από όλες τις προσπάθειες είναι σχετικό με την εύρεση των ισοδύναμων εκδόσεων. Αυτό είναι κάτι που στην πλειοψηφία των προσεγγίσεων παραμένει στην ευθύνη του ελεγκτή, πέρα από την περίπτωση του Mothra, ενός συστήματος για προγράμματα σε Fortran, το οποίο είναι αποτέλεσμα μακρόχρονων μελετών. Άλλες σημαντικές προσεγγίσεις είναι αυτή της «αδύναμης» (weak mutation) [How82], της «συμπαγούς» (firm mutation) [WH88] που βρίσκεται ανάμεσα στην προηγούμενη και στην «ισχυρή» προσέγγιση (strong mutation), αυτής που χρησιμοποιεί σχήματα [UOH93] και της επιλεκτικής (selective) [VM98] που εφαρμόζει τους τελεστές στους οποίους οφείλεται η πλειοψηφία των μεταλλάξεων. Επίσης τα τελευταία χρόνια με την εμφάνιση αντικειμενοστρεφών γλωσσών η μέθοδος έχει επεκταθεί για να εφαρμοστεί σύμφωνα με τα καινούρια χαρακτηριστικά τους. Μελετήθηκε η μετάλλαξη ολοκλήρωσης συστημάτων και

διεπαφών επικοινωνίας [DMM01, GM01, VMBD01], όπως και η μετάλλαξη κλάσεων [KCM00, KCM00b].

Η προτεινόμενη προσέγγιση

Μελετώντας λοιπόν την υπάρχουσα αρθρογραφία εντοπίστηκαν ισχυρά πλεονεκτήματα από διάφορες προσεγγίσεις, αλλά και μειονεκτήματα τους που έπρεπε όσο το δυνατό να περιοριστούν. Η αρχική ιδέα ήταν να δημιουργηθεί ένα μόνο υπερ-μεταλλαγμένο πρόγραμμα που θα περιείχε όλες τις μεταλλάξεις με τρόπο σχεδόν αντίστοιχο με αυτό που προτάθηκε από το MSG σύστημα (Mutant Schema Generator) [UOH93]. Για να γίνει αυτό θα πρέπει πριν τη φάση της μετάφρασης και χρησιμοποιώντας πληροφορία από την ανάλυση του προγράμματος να μετασχηματιστεί ο κώδικας φυτεύοντας κλήσεις σε στατικά προκαθορισμένες μεθόδους, αλλά και σε άλλες που θα δημιουργήσει κάποιος μετασχηματιστής δυναμικά. Ταυτόχρονα θα πρέπει να φυτεύονται κλήσεις σε μεθόδους με τρόπο που θα καθιστούν διαθέσιμα δεδομένα από την φάση της μετάφρασης και στη φάση της εκτέλεσης του κώδικα.

Η σύγκριση των αποτελεσμάτων των μεταλλαγμένων εκδόσεων σε σχέση με αυτά της πρωτότυπης αποτελεί ένα αρκετά δαπανηρό χρονικά βήμα της παραδοσιακής διαδικασίας ελέγχου με μεταλλάξεις, αναζητήθηκε τρόπος αυτόματου ελέγχου για τον καθορισμό των νεκρών ή ισοδύναμων εκδόσεων. Για να πραγματοποιηθεί αυτό έπρεπε να προστεθούν συγκριτικοί έλεγχοι σε κάποιο σημείο της ροής του ελεγχόμενου υπερ-μεταλλαγμένου προγράμματος. Σύμφωνα πάλι με την αρθρογραφία [How82] δεν είναι ανάγκη να εκτελεστεί μέχρι τέλους το πρόγραμμα αν ακολουθείται η «αδύναμη» προσέγγιση, αλλά μπορεί να εκτελεστούν όλες οι μεταλλάξεις για ένα σημείο την στιγμή που το πρόγραμμα φτάσει εκεί. Με βάση αυτή την ιδέα επεκτείναμε τα σχήματα με τρόπο τέτοιο, ώστε να εκτελούν όλες τις δυνατές μεταλλάξεις ενός σημείου μόλις φτάσει η εκτέλεση σε αυτό. Έπειτα μέσα από το ίδιο σχήμα να συγκρίνονται τα αποτελέσματα με αυτά που δίνει ο πρωτότυπος κώδικας.

Η αδύναμη όμως μετάλλαξη σύμφωνα με μελέτες έδινε λανθασμένες εκτιμήσεις για τα δεδομένα, μετρώντας μεγαλύτερους βαθμούς αξιοπιστίας τους. Για να μειωθεί αυτή η απόκλιση από την ισχυρή μετάλλαξη λήφθηκαν υπόψη οι σχετικές παρατηρήσεις [OL94, OLR+96]. Έτσι στην περίπτωση των επαναληπτικών δηλώσεων προτιμήθηκε να εφαρμοστεί η συμπαγής μετάλλαξη (firm mutation)

[WH88], πάντα μέσα σε σχήματα και εξετάζοντας τις αλλαγές τον συμμετεχόντων μεταβλητών μετά από Ν κύκλους. Σε αυτό τον έλεγχο βοηθάει ο μετασχηματισμός σε κλάση του κύκλου και η δημιουργία μεθόδου που θα ελέγχει όλες οι μεταβλητές της.

Επίσης δεν εφαρμόστηκαν τελεστές μετάλλαξης, που να αλλάζουν την ροή του προγράμματος, αφού μας ενδιέφερε η αδύναμη εκδοχή. Αυτό είναι κάτι που προτείνεται και από την επιλεκτική μετάλλαξη (selective mutation) [ORZ93, VM98, MB99]. Άλλοι τελεστές που δεν εφαρμόστηκαν ενώ υπήρχαν σε προηγούμενα συστήματα για άλλες γλώσσες ήταν λόγω των διαφορών της JAVA με αυτές, αλλά και επειδή ενισχύθηκε η υπόθεση για την ικανότητα του προγραμματιστή. Ένα σύστημα ελέγχου με την μέθοδο των μεταλλάξεων είναι υπολογιστικά και χρονικά ακριβό. Αυτό σημαίνει ότι προορίζεται για να ελέγξει κρίσιμα προγράμματα όσο αφορά θέματα ασφαλείας. Κατά συνέπεια είναι προφανές ότι σε τέτοια προγράμματα δεν πρόκειται να δουλεύει ένας αρχάριος προγραμματιστής. Έτσι κάποια λάθη που «φυτεύονταν» από άλλα συστήματα θεωρήθηκαν απίθανα και δεν συμπεριλήφθηκαν στις προδιαγραφές του παρόντος.

Τέλος η όλη προσέγγιση ενώ δεν είχε αρχικά σχεδιαστεί με σκοπό να εκμεταλλευτεί την ροή του προγράμματος για την δημιουργία των μεταλλάξεων απέδειξε το αντίθετο. Έχοντας «φυτέψει» στο πρόγραμμα σχήματα που περιέχουν τις μεταλλάξεις δεν σημαίνει ότι αυτές έχουν δημιουργηθεί κιόλας. Η δημιουργία τους συμβαίνει την στιγμή της εκτέλεσης του σημείου που αυτές μεταλλάσσουν. Αν τα δεδομένα ελέγχου δεν είναι ικανά να φτάσουν σε κάποιο σημείο του κώδικα τότε δεν προστίθενται στο γενικότερο σύνολο μεταλλάξεις για εκείνο το σημείο. Έτσι με την οδηγούμενη από τα δεδομένα δημιουργία μεταλλάξεων, αποφεύγεται η μετάλλαξη νεκρού κώδικα που ανήκει σε απροσπέλαστα μονοπάτια. Έμμεσα δηλαδή γίνεται μια ανάλυση κάλυψης των μονοπατιών του προγράμματος και έπειτα δημιουργούνται οι όποιες διαφορετικές εκδόσεις.

Συμπεράσματα

Σύμφωνα με παλιότερες μελέτες το μέγιστο πλήθος των αναμενόμενων μεταλλάξεων για ένα πρόγραμμα είναι όσο το τετράγωνο των αναφορών σε μεταβλητές. Για το προτεινόμενο σύστημα δεν είναι εύκολο να βρεθεί με κάποιο τρόπο το ακριβές πλήθος των μεταλλάξεων. Με βάση όμως την περιγραφή των

προδιαγραφών του και τον τρόπο λειτουργίας των τελεστών μετάλλαξης που θα εφαρμόζει μπορεί να εκτιμηθεί το μέγιστο αναμενόμενο από τον παρακάτω τύπο:

$$\sum_{t=1}^{T}(VR_t \cdot (V_t - 1)) + \sum_{l=0}^{L}\left[\sum_{t=1}^{T}(IV_t \cdot V_t) + 3 \cdot IC\right]_l + 8 \cdot L + \sum_{s=0}^{S}(2 \cdot C_s + 1) + 3 \cdot CR + 4 \cdot AOR + 2 \cdot LOR + 2 \cdot BOR + 5 \cdot ROR$$

Μεταβλητή	Εξήγηση
VR_t	Αναφορές σε μεταβλητές τύπου t
V_t	Μεταβλητές τύπου t ή τύπου επιτρεπτής μετάπτωσης σε t (type casting)
T	Διαφορετικοί τύποι μεταβλητών στο πρόγραμμα.
IV_t	Πλήθος αρχικοποιήσεων με μεταβλητές τύπου t.
IC	Πλήθος αρχικοποιήσεων με σταθερές τιμές.
L	Πλήθος επαναληπτικών δηλώσεων (while, for, do – while)
S	Πλήθος switch δηλώσεων
C_s	Πλήθος case περιπτώσεων σε κάθε switch-case δήλωση s
CR	Πλήθος εμφανιζόμενων σταθερών μεταβλητών ή τιμών.
AOR	Πλήθος εμφανιζόμενων αριθμητικών τελεστών.
LOR	Πλήθος εμφανιζόμενων λογικών τελεστών.
BOR	Πλήθος εμφανιζόμενων δυαδικών (binary) τελεστών.
ROR	Πλήθος εμφανιζόμενων σχεσιακών τελεστών.

Προοπτικές

Όσο αφορά τις μελλοντικές επεκτάσεις της προτεινόμενης προσέγγισης, αρχικά όπως προαναφέρθηκε θα πρέπει να μελετηθούν τα χαρακτηριστικά της αντικειμενοστρέφειας της JAVA και να εφαρμοστούν σχετικοί τελεστές ή ακόμα και να προταθεί υποστήριξη συγκεκριμένων χαρακτηριστικών από την γλώσσα που θα διευκολύνουν τον έλεγχο με μεταλλάξεις. Έπειτα θα ήταν ίσως χρήσιμο να μελετηθεί κατά πόσο θα μπορούσαν να προσαρμοστούν μέσα στα σχήματα αναλυτικές μέθοδοι και να βοηθήσουν στον έλεγχο ισοδυναμίας των ελεγχόμενων εκδόσεων. Στην αρθρογραφία γίνεται λόγος για ελέγχους βάση περιορισμών (CBT – Constraint-Based

Testing) [OP97]. Κάτι τέτοιο ίσως να βοηθούσε στην αυτοματοποιημένη αξιολόγηση πιθανών ισοδύναμων μεταλλάξεων αν εφαρμοζόταν μέσα στο κάθε σχήμα.

Επίσης η χρήση μηχανικής μάθησης ίσως θα μπορούσε να οδηγήσει σε δυναμικό καθορισμό των ποιων τελεστών μετάλλαξης θα έπρεπε να χρησιμοποιεί το σύστημα σύμφωνα με τις συνήθειες των προγραμματιστών και την κρισιμότητα του προγράμματος που ελέγχεται. Αυτό ίσως να καταργούσε τα στατικά δημιουργημένα σχήματα και να ανέθετε στο μετασχηματιστή την ευθύνη του καθορισμού τους μαζί με τα υπόλοιπα. Ακόμα θα ήταν χρήσιμο να ελεγχθεί κατά πόσο η τεχνική του τεμαχισμού προγραμμάτων (program slicing) [OC94, Wei84, HHD99] θα βοηθούσε σε απλούστερα προγράμματα και κατά συνέπεια μια επιπλέον μείωση του πλήθους των μεταλλάξεων.

Ευχαριστίες

Θερμές ευχαριστίες στους γονείς μου, που στήριξαν με κάθε τρόπο όλες τις προσπάθειες μου μέχρι το πέρας της παρούσας διπλωματικής εργασίας. Εκφράζω επίσης τις ευχαριστίες μου προς τον αναπληρωτή καθηγητή του τμήματος Πληροφορικής του Οικονομικού Πανεπιστημίου Αθηνών, κ. Εμμ. Γιακουμάκη, ο οποίος ήταν εξωτερικός κριτής. Τέλος ευχαριστώ τον επιβλέποντα της εργασίας αυτής, επίκουρο καθηγητή στο τμήμα Πληροφορικής του Οικονομικού Πανεπιστημίου Αθηνών, κ. Νικόλαο Μαλεύρη, για την βοήθεια και τις παρατηρήσεις του καθ' όλη την διάρκεια της εκπόνησης της μελέτης μου.

Κεφάλαιο 1ο: Εισαγωγικές έννοιες ελέγχου λογισμικού

Σε αυτό το πρώτο κεφάλαιο παρουσιάζονται κάποιες σημαντικές έννοιες στην περιοχή του ελέγχου λογισμικού. Ο έλεγχος λογισμικού, το σωστό πρόγραμμα, ο έλεγχος μονάδας προγράμματος, ο έλεγχος με εμφύτευση λαθών και η περίπτωση ελέγχου είναι έννοιες που σχετίζονται με την μέθοδο ελέγχου με μεταλλάξεις. Επίσης, είναι ιδιαίτερα διαδεδομένες στην βιβλιογραφία και συναντώνται συχνά και στην εργασία που ακολουθεί, για αυτό και δίνονται κάποιες σύντομες επεξηγήσεις τους για καλύτερη παρακολούθηση.

1.1. Έλεγχος λογισμικού

Ο έλεγχος (λογισμικού) είναι μια σημαντική διαδικασία στην προσπάθεια για παραγωγή ενός όσο το δυνατό καλύτερου προϊόντος λογισμικού. Υπάρχουν αρκετές μέθοδοι και τεχνικές που εφαρμόζονται για έλεγχο λογισμικού, από απλές ανιχνεύσεις διαρροών μνήμης μέχρι την κάλυψη πολύπλοκων συνθηκών. Άλλοτε ο στόχος είναι να μετρηθεί και να αξιολογηθεί η αξιοπιστία του προγράμματος, ενώ σε άλλες περιπτώσεις σκοπός είναι η βελτίωση του με την εύρεση και διόρθωση των υπαρχόντων λαθών. Ο συνδυασμός των δύο προσεγγίσεων έχει οδηγήσει σε μια τρίτη κατηγορία ελέγχου η οποία μετράει την βελτίωση της αξιοπιστίας ενός προγράμματος μετά από διόρθωση των λαθών που ανακαλύφθηκαν. Ο έλεγχος αυξανόμενης αξιοπιστίας (reliability growth testing) βεβαιώνει ότι η διόρθωση λαθών δεν δημιουργεί καινούρια.

Παρόλα αυτά ο έλεγχος λογισμικού στην γενικότητα του αντιμετωπίζει κάποια άλυτα προβλήματα. Παράγοντες για αυτά τα προβλήματα είναι η αδυναμία να αποφασιστεί πότε ένα πρόγραμμα στην διάρκεια του ελέγχου πρέπει να σταματήσει την εκτέλεση του (αν εκτελεί μια περίπτωση που το θέτει σε ατελείωτο κύκλο). Επίσης, δεν είναι δυνατός ο εξαντλητικός έλεγχος για αρκετά προγράμματα λόγω της αυξημένης πολυπλοκότητας τους, που απαιτεί εξωπραγματικό χρόνο για εξάντληση όλων των πιθανών συνδυασμών περιπτώσεων. Τέλος για να κρίνεται ένα λογισμικό σχετικά με το κατά πόσο εκτελεί σωστά τη δουλεία του και παράγει τα ακριβή αποτελέσματα θα πρέπει να είναι γνωστό από πριν το αποτέλεσμα, κάτι που αν ήταν δυνατό θα καθιστούσε περιττό το ίδιο το πρόγραμμα.

Έτσι το όλο πρόβλημα στην περίπτωση της προσπάθειας βελτίωσης του λογισμικού ανάγεται στην σύντομη και συγχρόνως αποτελεσματικότερη δημιουργία δεδομένων και σεναρίων ελέγχου, τα οποία θα καλύπτουν τις πιο κρίσιμες περιπτώσεις και σε σύντομο χρονικό διάστημα θα αποκαλύπτουν τα τυχόν προβλήματα. Μάλιστα αυτό πρέπει να γίνει με τρόπο που να μην χρειάζεται να εξεταστούν όλα τα πιθανά δεδομένα για να βρεθούν τα καλύτερα κάτι που θα ήταν ανέφικτο χρονικά.

1.2. Σωστό πρόγραμμα

Σωστή έκδοση κάποιου προγράμματος φαίνεται να θεωρείται αυτή που κυκλοφορεί για πραγματική χρήση (Release version) και δεν παρουσιάζει προβλήματα σχετικά με την σταθερότητα, την λειτουργικότητα και τα αποτελέσματα εξόδου από την επεξεργασία των εισαγομένων δεδομένων. Δεν υπάρχει ορισμός του σωστού ή λάθους προγράμματος στην Τεχνολογία Λογισμικού, γιατί αυτό προϋποθέτει την γνώση του αποτελέσματος του υπολογισμού που τελεί το πρόγραμμα. Απλά η μέθοδος θεωρεί ως σωστή την έκδοση του προγράμματος η οποία έχει τεθεί σε πραγματικές συνθήκες χρήσης. Έτσι η παρουσία και η ερμηνεία αυτού του όρου έχει αξία μόνο στα πλαίσια του ελέγχου κάποιου προγράμματος με μεθόδους εμφύτευσης λαθών, μια κατηγορία των οποίων είναι και αυτή των μεταλλάξεων.

1.3. Έλεγχος μονάδας προγράμματος (unit testing)

Συχνά και ιδιαίτερα σήμερα λόγω μεγέθους ένα λογισμικό σύστημα είναι αδύνατο να ελεγχθεί σαν σύνολο στο όλο του. Για αυτό το λόγο τεμαχίζεται σε μικρότερα μέρη, ώστε να μπορεί να είναι ευκολότερα ελέγξιμο από την ανθρώπινη αντίληψη, αλλά και υπολογιστικά και χρονικά ολοκληρώσιμο. Συχνά αυτός ο τεμαχισμός μπορεί να είναι έμμεσα προτεινόμενος από τα υποσυστήματα χάρη στο σχεδιασμό του συνολικού συστήματος, ενώ άλλες φορές πρέπει να εφαρμοστεί με διάφορες μεθόδους όπως το program slicing όπως αναφέρεται στην βιβλιογραφία (τεμαχισμός προγράμματος). Ιδιαίτερα σήμερα πάντως οι αντικειμενοστρεφής γλώσσες και η βαθμωτή αρχιτεκτονική (modular architecture) διευκολύνουν αυτό το τεμαχισμό σε μονάδες που μπορούν να ελεγχθούν αυτόνομα.

1.4. Περίπτωση ελέγχου (Test case)

Είναι δεδομένα ελέγχου ικανά να αποκαλύψουν ένα λάθος στο πρόγραμμα [OC94, Won93]. Για να χαρακτηριστεί επαρκές ένα σύνολο δεδομένων ελέγχου πρέπει το «σωστό» πρόγραμμα να συμπεριφέρεται σωστά για κάθε σενάριο από το σύνολο, ενώ τα «λάθος» προγράμματα να συμπεριφέρονται λανθασμένα έστω για ένα σενάριο από το σύνολο αυτών [Mar89]. Τεχνικές που χρησιμοποιούνται για την δημιουργία και κυρίως την αξιολόγηση test cases, ώστε να ικανοποιούν συγκεκριμένα κριτήρια είναι [KO91]:

- **Κάλυψη Μονοπατιών - Feasible Path Coverage/Analysis (FPA)**

Είναι μια στην βάση της αναλυτική μέθοδος που εξετάζει ποια μονοπάτια δεν πρόκειται να εκτελεστούν. Εκτελεί και δοκιμάζει με δεδομένα το πρόγραμμα για να καταλήξει τελικά στο κατά πόσο είναι ικανά να καλύψουν ή όχι όλα τα μονοπάτια του. Το πρόβλημα που προσπαθεί να απαντήσει αναφέρεται στην βιβλιογραφία ως Feasible Test Problem (FTP). Κάτι που σχετίζεται άμεσα με την κάλυψη των μονοπατιών είναι και η κάλυψη των δηλώσεων (statement coverage testing). Αυτό που γίνεται με αυτή την μέθοδο είναι η κατηγοριοποίηση των δεδομένων ή αλλιώς των περιπτώσεων ελέγχου (test cases), έτσι ώστε αν επιλεγεί κάποιο από μια συγκεκριμένη κατηγορία, τότε να είναι σίγουρο ότι η εκτέλεση του προγράμματος θα περάσει από μια συγκεκριμένη δήλωση (statement).

Οι τεχνικές για έλεγχο μονοπατιών διακρίνονται σε αυτές που στηρίζονται στην ροή ελέγχου (control-flow driven) και σε αυτές που εξετάζουν τη ροή δεδομένων (data-flow driven) [Woo88]. Στην πρώτη κατηγορία άλλοτε ελέγχεται το κατά πόσο τα δεδομένα είναι ικανά να εκτελέσουν κάθε δήλωση του προγράμματος (statement testing) ή αν επιτυγχάνουν την εκτέλεση από μια έστω φορά κάθε κλάδου του. Όσο για τις τεχνικές που βασίζονται σε ροή δεδομένων ο σκοπός είναι να δημιουργηθούν δεδομένα ελέγχου που να πετυχαίνουν διάφορους συνδυασμούς εκτέλεσης κώδικα μεταξύ ορισμών μεταβλητών (variable definition) και αναφορών (variable reference) σε αυτές.

- **Συμβολικός Έλεγχος - Symbolic testing**

Η συμβολική εκτέλεση είναι μια μέθοδος που φροντίζει να δημιουργήσει λογικές συνθήκες από την ανάλυση της δομής του προγράμματος και έπειτα να προσπαθήσει να ελέγξει ποιες περιπτώσεις τις ικανοποιούν ή όχι. Είναι αναλυτική μέθοδος και δεν φαίνεται να χρησιμοποιείται ιδιαίτερα σε συστήματα. Η λογική της είναι να δημιουργηθούν συναρτήσεις και γνωρίζοντας το επιθυμητό αποτέλεσμα ικανοποίησης τους ή όχι να γίνει προσπάθεια εντοπισμού των αναγκαίων δεδομένων ελέγχου.

- **Λειτουργικός Έλεγχος - Functional testing**

Ο λειτουργικός έλεγχος γίνεται για να επιβεβαιωθεί η σταθερότητα του προγράμματος και η αναμενόμενη συμπεριφορά του σύμφωνα με τις προδιαγραφές. Αν και ελέγχει τα αποτελέσματα κυρίως επικεντρώνεται στην ροή του προγράμματος. Αυτό που γίνεται είναι να δοκιμάζεται με βάση κάποια σενάρια χρήσης το σύστημα και να παρατηρεί ο ελεγκτής τις αποκρίσεις και την γενικότερη συμπεριφορά του. Το ζητούμενο είναι να ικανοποιούνται οι προδιαγραφές με τις οποίες σχεδιάστηκε και υλοποιήθηκε το ελεγχόμενο σύστημα. Επίσης σε αυτό το στάδιο ελέγχονται και πιο αφηρημένες έννοιες όπως η ευχρηστία.

- **Ανάλυση Μεταλλάξεων - Mutation analysis**

Είναι μια περίπτωση εφαρμογής fault-based testing. Πηγαίνει όμως ένα βήμα πιο μπροστά από τις υπόλοιπες αφού παρέχει πληροφορία ακόμα και αν δεν υπάρχουν λάθη στο πρόγραμμα. Οι υπόλοιπες τεχνικές αυτό που κάνουν είναι να αναζητούν να βρουν τα λάθη που θα υπάρχουν πιθανών στο υπό εξέταση πρόγραμμα. Περισσότερα και αναλυτικότερα για την συγκεκριμένη μέθοδο αναφέρονται στο επόμενο κεφάλαιο.

1.5. Τεχνικές «άσπρου» ή «μαύρου κουτιού» (white/ black-box testing)

Πριν αναφερθούν οι τεχνικές που βασίζονται σε λάθη κατά τον έλεγχο του προγράμματος είναι ωφέλιμο να αναφερθούν οι δύο βασικές κατηγορίες ελέγχου προγραμμάτων, αυτή του «μαύρου κουτιού» (black-box testing) και αυτή του «άσπρου κουτιού» (white-box testing). Στην πρώτη το πρόγραμμα ελέγχεται με τυχαία δεδομένα και απαιτείται ένας παρατηρητής (oracle) να περνάει δεδομένα στο

πρόγραμμα και έπειτα να ελέγχει τα αποτελέσματα. Ο ελεγκτής αυτός δεν μπορεί να γνωρίζει οτιδήποτε σχετικά με την δομή του λογισμικού που ελέγχει. Ο έλεγχος προδιαγραφών (specification-based testing) και ο αντίστοιχος για τις απαιτήσεις (requirements-based testing) είναι κατηγορίες ελέγχου που εφαρμόζουν την τεχνική του «μαύρου κουτιού». Από την άλλη υπάρχει η τεχνική του «άσπρου κουτιού» η οποία εξετάζει τον κώδικα και ελέγχει την λογική της δομής του. Βέβαια σήμερα με τα υπερμεγέθη, από άποψη κώδικα, λογισμικά συστήματα δεν είναι δυνατό ο έλεγχος αυτός να γίνει στο σύνολο του συστήματος, εφαρμόζεται όμως σε υποσυστήματα του.

1.6. Τεχνικές ελέγχου βασισμένες σε λάθη (fault-based testing)

Πριν ακόμα προταθεί η μέθοδος των μεταλλάξεων υπήρχε η τεχνική της «σποράς λαθών» (fault seeding) στον κώδικα. Είχε προταθεί το 1972 από τον Harlan Mills και είχε σκοπό την μέτρηση της αξιοπιστίας του προγράμματος. Αυτό που έκανε ήταν να προσθέτει (να «εμφυτεύει») γνωστά λάθη στον κώδικα και έπειτα να ελέγχει το πρόγραμμα.. Παρακολουθώντας το πλήθος των λαθών που εμφυτεύτηκαν και ανακαλύφθηκαν όπως και αυτών που βρέθηκαν ενώ ήδη υπήρχαν μπορεί να υπολογιστεί με διάφορους αλγορίθμους το υπολειπόμενο πλήθος των λαθών στον κώδικα.

Ο εμβολιασμός λαθών (fault injection) στην πιο κλασσική του υλοποίηση ανήκει στην κατηγορία του «άσπρου κουτιού», αφού η προσθήκη λαθών και η μετάλλαξη του κώδικα του προγράμματος σημαίνει χρήση του. Επιπλέον απαιτεί την διαθεσιμότητα του κώδικα και τον δομών δεδομένων. Βέβαια υπάρχει και εκδοχή της τεχνικής σε επίπεδο δεδομένων λίγο πριν εισέλθουν στο πρόγραμμα για έλεγχο. Αυτό την κάνει να ανήκει τότε στην κατηγορία του «μαύρου κουτιού», αφού δεν εξετάζεται του πρόγραμμα εσωτερικά.

Γενικά οι δύο παραπάνω τεχνικές μοιάζουν αρκετά μεταξύ τους αλλά και με τη μέθοδο ελέγχου με μεταλλάξεις, η οποία παρουσιάστηκε χρονικά κάπου ενδιάμεσα των δύο. Ο έλεγχος βασισμένος σε λάθη (fault-based testing) γενικότερα είναι μια στρατηγική ελέγχου που αποσκοπεί στην δημιουργία δεδομένων ελέγχου ταξινομημένα σε ομάδες που διακρίνονται για την ικανότητα τους να αποκαλύψουν ένα είδος λαθών κάθε φορά [OC94]. Κατά συνέπεια χρησιμοποιείται για την αξιολόγηση των δεδομένων ελέγχου.

Κεφάλαιο 2ο: Μέθοδος μεταλλάξεων (mutation testing/ analysis)

Η μέθοδος των μεταλλάξεων (mutation testing) είναι μια διαδικασία διαφοροποίησης του πηγαίου κώδικα με σκοπό να εντοπιστούν ασάφειες που μπορεί να υπάρχουν σε αυτόν. Αυτές οι ασάφειες μπορεί να προκαλέσουν αποτυχίες στο λογισμικό αν δεν ανιχνευτούν και δεν διορθωθούν εγκαίρως από τον προγραμματιστή. Επειδή αυτά τα λάθη είναι συνήθως δυσδιάκριτα και μη ανιχνεύσιμα κατά τη διάρκεια των κλασικών προγραμματιστικών ελέγχων, γίνονται συνήθως αντιληπτά από τον πελάτη όταν το πρόγραμμα έχει περάσει στα χέρια του. Σκοπός λοιπόν αυτής της μεθόδου είναι η ανίχνευση αυτών των λαθών και η έγκαιρη διόρθωση τους. Όπως αναφέρει και ο Woodward [Woo88] η μέθοδος αυτή φαίνεται να λειτουργεί ενισχυτικά για το σχόλιο του Dijkstra ότι «ο έλεγχος αναδεικνύει την ύπαρξη λαθών (bugs) και ποτέ την απουσία τους».

Σε κάποια άλλη εργασία [VMBD01] αναφέρεται ότι στόχος του ελέγχου με χρήση μεταλλάξεων είναι να ενθαρρυνθεί ο ελεγκτής για να βρει σενάρια ελέγχου που θα οδηγήσουν σε διαφορετικά αποτελέσματα τις μεταλλάξεις. Από το σύνολο τις αρθρογραφίας μάλιστα δίνεται η εντύπωση ότι σε πρακτικό επίπεδο η μέθοδος μπορεί να χρησιμοποιηθεί σαν τρόπος αξιολόγησης των δεδομένων ελέγχου και σαν μέτρο της αξιοπιστίας τους. Αυτό που μετράται είναι το κατά πόσο είναι ικανά τα υπάρχοντα δεδομένα να ανακαλύψουν αλλαγές στον κώδικα, οι οποίες με την σειρά τους οδηγούν σε διαφορετικά αποτελέσματα κάποια μονάδα προγράμματος. Αυτό βέβαια σημαίνει ότι κάποιο μη αναμενόμενο αποτέλεσμα θα οδηγούσε σε διόρθωση του κώδικα για την παραγωγή του αντίστοιχου επιθυμητού. Κατά συνέπεια είναι και μια μέθοδος που βοηθάει στην διόρθωση λαθών σε προγράμματα στις περιπτώσεις που γνωρίζουμε το επιθυμητό αποτέλεσμα. Αυτό θα γινόταν θέτοντας κάποια μετάλλαξη που θα έδινε το επιθυμητό αποτέλεσμα ως την σωστή έκδοση προγράμματος. (Αυτό ίσως να σημαίνει ότι η μέθοδος θα μπορούσε σε κάποια υλοποίηση της να αναζητεί και την εύρεση του mutant που δίνει συγκεκριμένο αποτέλεσμα.)

2.1. Ορισμός και ορολογία

Η μέθοδος των μεταλλάξεων [DLS78] είναι μια ισχυρή τεχνική που βασίζεται στην εμφύτευση λαθών στον κώδικα του υπό εξέταση λογισμικού σε επίπεδο μονάδας προγράμματος (program unit testing). Βοηθάει τον ελεγκτή-χρήστη να δημιουργήσει και να ελέγξει **δεδομένα ελέγχου (test data)** με επαναληπτικό (iteratively) και αλληλεπιδραστικό τρόπο για ενίσχυση της αποτελεσματικότητας τους [OC94]. Ο υπεύθυνος του ελέγχου είναι αυτός που καθορίζει το είδος και το πλήθος των τελεστών μετάλλαξης που θα χρησιμοποιηθούν, γνωρίζοντας ποια είναι πιο κρίσιμα για το κάθε σύστημα [Mar89]. Ανήκει στην κατηγορία των στρατηγικών ελέγχου με βάση τα λάθη **(fault-based testing strategies)** με την διαφορά ότι επικεντρώνεται στην **«μετάλλαξη» (mutation)** του κώδικα δημιουργώντας συντακτικά λάθη. Η μετάλλαξη αφορά λάθη που μπορεί να υπάρχουν στην εφαρμογή κάποιου τελεστή (operator), τελεσταίου (operand) ή στοιχείο ελέγχου (control element) [Mar89]. Έτσι δημιουργούνται πολλές διαφοροποιημένες εκδόσεις του πηγαίου κώδικα, γνωστές στην ορολογία ως **«μεταλλάξεις» ή «μεταλλαγμένες» εκδόσεις (mutants)** του αρχικού προγράμματος, με σκοπό να δοκιμαστούν με δεδομένα και να αποτύχουν.

Σύμφωνα με την ορολογία της μεθόδου αυτό γίνεται από ένα **τελεστή μετάλλαξης (mutation operator)**. Οι τελεστές αυτοί είναι αυστηρά εξαρτώμενοι από την γλώσσα προγραμματισμού (language-dependent). Αυτό σημαίνει πως σε δύο διαφορετικές γλώσσες μπορούν να διαφέρουν από τον τρόπο σύνταξης καθεμίας ως και την ίδια την λειτουργικότητα τους. Σε περιπτώσεις που το λογισμικό είναι γραμμένο σε κάποια αντικειμενοστρεφή γλώσσα προγραμματισμού (π.χ. Java, C++) η μέθοδος των μεταλλάξεων γίνεται πιο σύνθετη εξαιτίας επιπλέον δυνατών μεταλλάξεων που μπορούν να παραχθούν από το αρχικό πρόγραμμα εξαιτίας των δυνατοτήτων που τους παρέχει ο αντικειμενοστρεφής χαρακτήρας της γλώσσας. Οι μεταλλάξεις που προκύπτουν από τον ίδιο τελεστή ανήκουν στον ίδιο **τύπο (type).**

Αποτυχία (failure) στην περίπτωση του ελέγχου με την μέθοδο των μεταλλάξεων σημαίνει ότι το αποτέλεσμα είναι διαφορετικό του αναμενόμενου με βάση την έξοδο του πρωτότυπου προγράμματος. Αν σημειωθεί αποτυχία στον έλεγχο του «μεταλλαγμένου» προγράμματος τότε θεωρείται **«σκοτωμένο» (killed) ή «νεκρό» (dead)** από την μέθοδο, ενώ τα δεδομένα ελέγχου κρίνονται ικανά να αποκαλύψουν τυχών λάθη στο πρωτότυπο πρόγραμμα. Αν τα δεδομένα δεν είναι

ικανά να ανακαλύψουν το λάθος στον κώδικα και κατά συνέπεια να «σκοτώσουν» κάποια μετάλλαξη, τότε αυτή αποτελεί ένα πιθανό **ισοδύναμο (equivalent mutant)** με το πρωτότυπο πρόγραμμα.

Αυτό αποτελεί και σημαντικό πρόβλημα γιατί είναι κάτι το οποίο δεν μπορεί να αποδειχτεί πάντα αυτοματοποιημένα, αφού πρέπει να ελεγχθεί η λογική σχεδίαση του προγράμματος στο σημείο της πιθανής ισοδυναμίας. Αποδεικνύεται ότι υπάρχει σχέση μεταξύ ισοδυναμίας και δημιουργίας δεδομένων ελέγχου από τους Budd και Angluin. «Αν υπάρχει υπολογίσιμη συνάρτηση που να ελέγχει την ισοδυναμία δύο προγραμμάτων τότε υπάρχει και μια άλλη υπολογίσιμη συνάρτηση για την δημιουργία επαρκών (adequate) δεδομένων ελέγχου ενός προγράμματος και το αντίστροφο» [BA82]. Στην πράξη είναι πολύ δύσκολο να δημιουργηθούν δεδομένα που να ανακαλύπτουν όλες τις δυνατές μεταλλάξεις [Mar89]. Η ποιότητα των δεδομένων ελέγχου, άρα και η αξιοπιστία τους, κρίνεται από το πλήθος των μη ισοδύναμων μεταλλάξεων ως προς το συνολικό πλήθος αυτών που δημιουργούνται (στην αρθρογραφία αναφέρεται ως Mutation Score - MS). Με το πλήθος των μη ισοδύναμων «μεταλλάξεων» εννοούνται οι «σκοτωμένες» συν το πλήθος από τις υπόλοιπες που εξετάστηκαν και βρέθηκε ότι δεν αποτελούν ισοδύναμες εκδόσεις του πρωτότυπου προγράμματος.

Αξιοπιστία δεδομένων = Μη ισοδύναμες «μεταλλάξεις» / Συνολικό πλήθος

Υπάρχει και ένας άλλος ορισμός για το βαθμό αξιοπιστίας της μεθόδου των μεταλλάξεων και αναφέρεται ως «βαθμός καταλληλότητας μετάλλαξης» (Mutation Adequacy Score και συμβολίζεται πάλι με MS). Αυτός δίνεται από τον παρακάτω τύπο [UOH93]:

$$MS_G(P,T) = \frac{\#Dead}{\#Mutants - \#Equivalent} \times 100\%$$

όπου P είναι το πρόγραμμα για το οποίο θεωρούμε ότι έχουμε σωστά αποτελέσματα χρησιμοποιώντας το σενάριο ελέγχου (test case) T. Το κλάσμα αυτό αλλού αναφέρεται και ως «καταλληλότητα δεδομένων ελέγχου» (Test Data Adequacy) [WHHR88]. Όταν το γινόμενο αυτό είναι ίσο με την μονάδα τότε λέγεται ότι το σενάριο T είναι επαρκές (adequate) σε σχέση με τον έλεγχο μεταλλάξεων.

[VMBD01]. Φαίνεται να υπολογίζει μεγαλύτερους συντελεστές, αφού στον παρανομαστή έχουν αφαιρεθεί οι ισοδύναμες μεταλλαγμένες εκδόσεις του πρωτότυπου προγράμματος, ενώ στις «σκοτωμένες μεταλλάξεις» δεν φαίνεται να περιλαμβάνονται και όσες δεν είναι ισοδύναμες. Με αυτές εννοούνται οι εκδόσεις που έχουν ίδιο αποτέλεσμα με το πρωτότυπο πρόγραμμα, αλλά διαφορετική αλγοριθμική λογική.

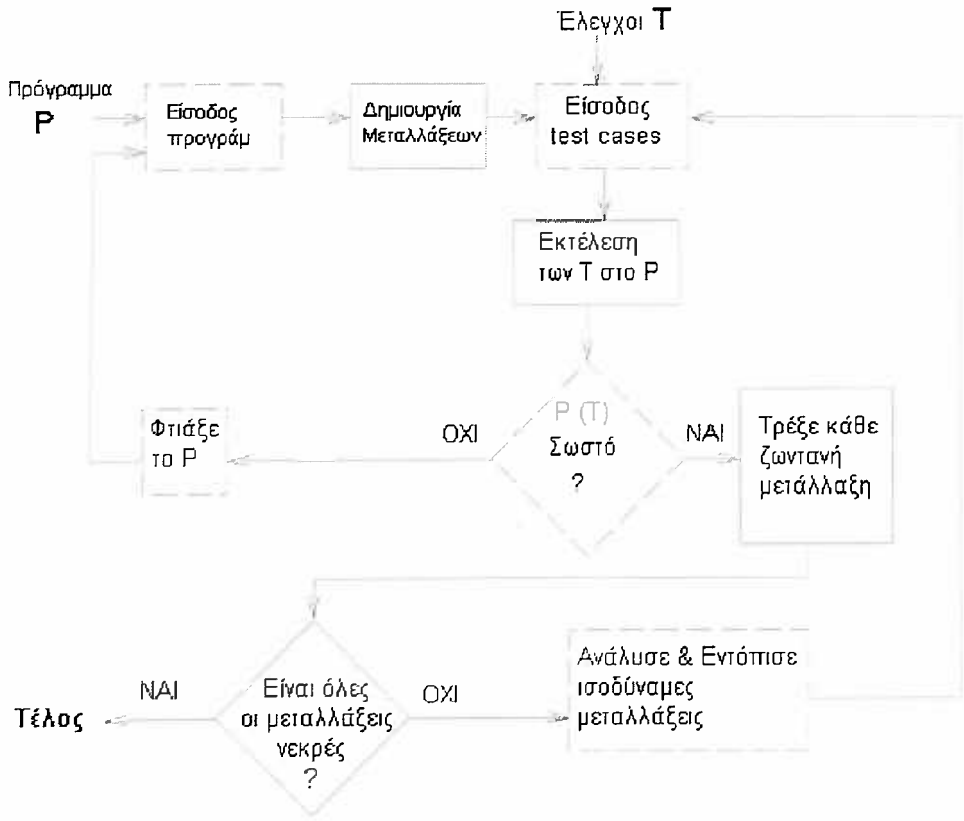
Από αυτό θα μπορούσε να θεωρηθεί ότι ο δεύτερος τρόπος υπολογισμού δίνει πιο αξιόπιστα νούμερα καταλληλότητας δεδομένων στην προσπάθεια αξιολόγησης τους με την μέθοδο των μεταλλάξεων. Τα δύο παραπάνω γινόμενα είναι ίσα στην περίπτωση που δεν υπάρχει κάποιος μηχανισμός αυτόματος ή μη που να αποφασίζει για την ισοδυναμία των ελεγχόμενων εκδόσεων. Σε αυτή την περίπτωση το πλήθος των ισοδύναμων εκδόσεων είναι μηδέν. Πειραματικά θα πρέπει να υπολογιστούν και οι δύο για να καταλήξουμε σε συμπεράσματα σχετικά και με την αξιοπιστία των παραπάνω συντελεστών.

2.2. Πώς λειτουργεί η μέθοδος;

Η διαδικασία ελέγχου με την μέθοδο των μεταλλάξεων ξεκινάει με την δημιουργία των μεταλλαγμένων εκδόσεων από το αυτοματοποιημένο σύστημα μετάλλαξης. Κλειδί για την επιτυχία της μεθόδου είναι η γνώση ή η πρόβλεψη των πιθανών λαθών που μπορεί να προκληθούν στην διάρκεια του κύκλου ανάπτυξης [Mar89]. Έπειτα προστίθενται σταδιακά οι περιπτώσεις ελέγχου (test cases) αυτοματοποιημένα ή μη και ο χρήστης-ελεγκτής του συστήματος είναι αυτός που ελέγχει την ορθότητα της εξόδου του πρωτότυπου προγράμματος σε κάθε περίπτωση. Αν δεν είναι η αναμενόμενη («σωστή»), τότε το πρόγραμμα πρέπει να διορθωθεί και να ξεκινήσει η διαδικασία από την αρχή, αλλιώς ελέγχονται οι μεταλλάξεις με τα δεδομένα κάθε περίπτωσης. Σε περίπτωση που το αποτέλεσμα μιας μετάλλαξης είναι διαφορετικό από αυτό του πρωτότυπου προγράμματος τότε «σκοτώνεται» και θεωρείται νεκρή από την μέθοδο για την συγκεκριμένη περίπτωση ελέγχου.

Αφού λοιπόν ελεγχθούν όλες οι περιπτώσεις για κάθε μετάλλαξη, κάποιες μεταλλαγμένες εκδόσεις δεν θα είναι νεκρές. Αυτές είτε είναι δυνατό να σκοτωθούν (killable) με κάποια καινούρια δεδομένα, είτε είναι ισοδύναμες εκδόσεις του πρωτότυπου προγράμματος. Αν αποδειχτεί ότι μια μετάλλαξη ανήκει στην τελευταία περίπτωση, τότε δεν χρειάζεται να λαμβάνεται υπόψη από το σύστημα ελέγχου.

Στο σχήμα που ακολουθεί αναπαριστάται η κλασική διαδικασία της μεθόδου των μεταλλάξεων. Σενάρια ελέγχου (test cases) παρέχονται στο σύστημα ως δεδομένα εισόδου στο πρόγραμμα P. Κάθε test case εκτελείται στο αυθεντικό πρόγραμμα P και ελέγχεται αν το αποτέλεσμα είναι σωστό. Αν δεν είναι, αυτό σημαίνει πως υπάρχει λάθος (bug) στο πρόγραμμα και θα πρέπει να διορθωθεί πριν το test case επαναχρησιμοποιηθεί. Αν είναι σωστό, τα σενάρια ελέγχου εκτελούνται για κάθε μετάλλαξη του προγράμματος (mutant program). Αν το αποτέλεσμα από μία μετάλλαξη του προγράμματος διαφέρει από το αυθεντικό (σωστό) αποτέλεσμα, η μετάλλαξη αποκαλείται «σκοτωμένη» (dead).



Σχήμα 1: Παραδοσιακή μέθοδος ελέγχου μεταλλάξεων

Στο παραπάνω σχήμα όσα κουτιά είναι με διακεκομμένη γραμμή σημαίνει ότι είναι διαδικασίες που γίνονται από τον ελεγκτή και όχι με κάποιον αυτοματοποιημένο τρόπο από κάποιο λογισμικό ή υλικό σύστημα.

2.3. Παραδοχές μεθόδου

Το πλήθος των λανθασμένων εκδόσεων ενός προγράμματος μπορεί να είναι αρκετά μεγάλο και η διαδικασία ελέγχου κάθε μιας υπολογιστικά ασύμφορη. Παρόλα αυτά όμως η μέθοδος των μεταλλάξεων βασίζεται σε δύο σημαντικούς κανόνες [OL94] χάρη στους οποίους περιορίζεται αυτό το πλήθος:

- **Υπόθεση ικανότητας του προγραμματιστή**

Η υπόθεση αυτή (competent programmer hypothesis [ABD+79]) δέχεται ότι ένας ικανός προγραμματιστής γράφει προγράμματα τα οποία τείνουν να είναι σωστά. Ο κώδικας του μπορεί να μην είναι σωστός, αλλά παρόλα αυτά απέχει από την επιθυμητή «σωστή» έκδοση μόνο μερικά λάθη.

- **Επίδραση σύζευξης λαθών**

Αυτό που περιγράφεται από τον συγκεκριμένο κανόνα (coupling effect [DLS78]) είναι ότι αν τα δεδομένα ελέγχου είναι αρκετά ικανά ώστε να βρουν απλά λάθη, τότε είναι αρκετά ευαίσθητα για να αποκαλύψουν και πιο σύνθετα λάθη στο πρόγραμμα.

2.4. Πλήθος μεταλλάξεων

Οι παραπάνω κανόνες δεν είναι πάντα αληθείς, αλλά χρησιμοποιούνται ως ικανές ευρετικές οδηγίες (heuristic guidelines) για την σωστή εφαρμογή της μεθόδου των μεταλλάξεων κατά τον έλεγχο ενός προγράμματος [OL94]. Παρόλα αυτά όμως δεν καταφέρνουν ικανοποιητική μείωση των δημιουργούμενων προς έλεγχο εκδόσεων. Δυναμικά ο αριθμός των μεταλλάξεων για ένα πρόγραμμα είναι άπειρος, πόσο μάλλον όταν σε μια μεταλλαγμένη έκδοση υπάρχουν περισσότερα του ενός «λάθη» [VM98]. Η εκτίμηση του A.T. Acree στην διδακτορική του διατριβή [Acr80] ανέφερε N^2 μεταλλάξεις (με εμφύτευση ενός λάθους) για πρόγραμμα στο οποίο εμφανίζονται N αναφορές σε μεταβλητές. Το σύστημα Mothra, που σχεδιάστηκε για έλεγχο προγραμμάτων γραμμένα σε Fortran σύμφωνα με την μέθοδο των μεταλλάξεων, δημιούργησε 951 μεταλλαγμένες εκδόσεις για πρόγραμμα (ταξινόμησης τριγώνων) 30 γραμμών (παραπάνω δηλαδή από τον υπολογισμό του Budd, δηλαδή το τετράγωνο των γραμμών του προγράμματος, $30^2 = 900 < 951$) [DGK+88]. Είναι προφανές λοιπόν ότι η συγκεκριμένη μέθοδος ελέγχου, αν και έχει καταφέρει να περιορίσει το πλήθος των λανθασμένων εκδόσεων ενός προγράμματος, είναι ακόμα αρκετά δαπανηρή υπολογιστικά ακόμα και για μικρά και απλά

προγράμματα. Χρειάζεται λοιπόν να βρεθεί μια τεχνική επιτάχυνσης της όλης διαδικασίας ή να αναπτυχθεί μια μέθοδος για την αποκοπή της πληθώρας των υποψηφίων μεταλλάξεων, ώστε να προκύπτουν λίγες που να μπορούν να ελεγχθούν με δεδομένα.

2.5. Κλάσεις τελεστών μετάλλαξης

Σύμφωνα με τα όσα γράφουν οι King και Offutt στην παρουσίαση του Mothra [KO91] το συγκεκριμένο σύστημα υποστηρίζει 22 τελεστές οι οποίοι αποτελούν συνέπεια δεκαετούς μελέτης των πιο κοινών και διαδεδομένων λαθών που κάνουν οι προγραμματιστές. Στο σύστημα αυτό θεωρήθηκε η παρακάτω κατηγοριοποίηση των τελεστών αυτών:

- **Δηλωτική ανάλυση (Statement analysis - sal)**
  Αλλάζει κάθε statement με μια από τις εντολές TRAP, RETURN και CONTINUE. Αλλάζει το στόχο σε κάθε κλήση GOTO ή DO.
- **Αποφαντική ανάλυση (Predicate analysis - pda)**
  Παίρνει την απόλυτη τιμή και την αρνητική απόλυτη τιμή κάθε έκφρασης. Αντικαθίσταται κάθε αριθμητικός τελεστής από καθένα από τους υπόλοιπους και ανάλογα γίνονται για κάθε λογικό και σχεσιακό τελεστή. Εισαγωγή μονοσήμαντων τελεστών πριν από κάθε έκφραση. Τροποποιούνται οι τιμές των σταθερών παραμέτρων (constants) και κλήσεων ανάθεσης τιμών (DATA statements)
- **Συμπτωματική ανάλυση (Coincidental analysis - cca)**
  Αντικατάσταση κλιμακούμενων μεταβλητών (scalar variables), αναφορών σε πίνακες και σταθερών τιμών (constants) από άλλες κλιμακούμενες μεταβλητές, αναφορές πινάκων και σταθερές τιμές. Αντικατάσταση ονόματος πίνακα από όνομα κάποιου άλλου.

Σύμφωνα πάλι με τον Howden τα είδη των συστατικών είναι πέντε. Με τον όρο συστατικό φαίνεται να εννοείται ο τελεστής μετάλλαξης, χωρίς όμως να έχει οριστεί από τον ίδιο σαφώς η έννοια του.

1. Αναφορά μεταβλητής (Variable reference)
2. Ανάθεση μεταβλητής (Variable assignment)
3. Αριθμητική έκφραση (Arithmetic expression)

4. Σχεσιακή έκφραση (Relational expression)
5. Διτήμαντη έκφραση (Boolean expression)

Σε άλλο σύστημα που δημιουργήθηκε για εφαρμογή της μεθόδου σε προγράμματα γραμμένα σε Ada83 [VM98] αναφέρονται 65 τελεστές μετάλλαξης οι οποίοι χωρίζονται σε πέντε κατηγορίες:

1. 30 τελεστές μετάλλαξης για **αντικατάσταση τελεσταίων** (operands –μεταβλητές, σταθερές, αναφορές σε πίνακες, αναφορές σε εγγραφές και δείκτες).
2. 14 τελεστές για **δηλώσεις**.
3. 14 τελεστές για **εκφράσεις** (όπως απόλυτης τιμής, αριθμητικών/ λογικών/ σχεσιακών/ unary τελεστών, «στριφογύρισμα» πεδίου (domain twiddle) και κάποιες εξαιρέσεις).
4. 4 τελεστές για **κάλυψη** (ώστε να αναγκάζουν την κάλυψη τουλάχιστον μια φορά κάθε δήλωσης στο πρόγραμμα)
5. 3 τελεστές **ανάθεσης εργασιών προγράμματος (tasking)**.

2.6. Παράδειγμα εφαρμογής μεθόδου μετάλλαξης

Αν για παράδειγμα έχουμε ένα πρόγραμμα υπό εξέταση με την μέθοδο των μεταλλάξεων αυτό που πρέπει να κάνουμε για κάθε γραμμή του κώδικα είναι να «εμφυτεύσουμε» λάθη με τέτοιο τρόπο, ώστε να δημιουργήσουμε «λανθασμένες» εκδόσεις του. Έπειτα θα εκτελέσουμε το κάθε «νέο» με τα δεδομένα ελέγχου που διαθέτουμε. Στο παρακάτω παράδειγμα παρουσιάζονται μερικές μεταλλάξεις που μπορούν να αντικαταστήσουν μια λογική συνθήκη ελέγχου.

0) if (A < B) then … (πρωτότυπο)

---

1) if (A <= B) then … (μεταλλάξεις)
2) if (A > B) then …
3) if (A >= B) then …
4) if (A == B) then …
5) if (A != B) then …

Σχήμα 2: Πιθανές μεταλλάξεις μιας συγκριτικής έκφρασης

Αυτό που προτείνει η συγκεκριμένη μέθοδος είναι ότι για κάθε μια εκδοχή (1-5) πρέπει να δημιουργηθεί ένα καινούριο πρόγραμμα, το οποίο θα διαφέρει από το πρωτότυπο στην συγκεκριμένη γραμμή κώδικα, η οποία θα έχει αντικατασταθεί με μια από τις μεταλλάξεις κάθε φορά. Έτσι για κάθε μονάδα προγράμματος υπό έλεγχο πρέπει να δημιουργηθούν «λανθασμένες εκδόσεις» αντικαθιστώντας κάθε φορά **μια μόνο** γραμμή με την αντίστοιχο μεταλλαγμένο κώδικα. Στον ορισμό της μεθόδου τονίζεται η εμφύτευση **ενός και μόνο** λάθους σε κάθε μονάδα προγράμματος. – Βέβαια στην βιβλιογραφία υπάρχει και μια αναφορά [VM98] που υποστηρίζει ότι μπορούν να δημιουργηθούν εκδόσεις που θα περιέχουν παραπάνω του ενός λάθη δημιουργώντας πολύπλοκες μεταλλάξεις (complex mutants). –

Κεφάλαιο 3ο: Υπάρχουσες προσεγγίσεις

Στην αρθρογραφία εμφανίζονται αρκετές προσεγγίσεις για την υλοποίηση της μεθόδου του ελέγχου προγραμμάτων με μεταλλάξεις. Θεωρητικές ή εφαρμοσμένες σε συστήματα, αναφέρονται σε μετάλλαξη του κώδικα, της κατάστασης των δεδομένων και της κατάστασης της στατικής ή τρέχουσας μνήμης στο σύνολο ή σε μέρος αυτών. Το τελευταίο γίνεται για παράδειγμα με την αναγκαστική επαναφορά κάποιων μεταβλητών στην αρχική τους κατάσταση σε κάποιο σημείο του κώδικα. Οι σημαντικότερες από αυτές είναι:

- *Σύστημα ελέγχου προγραμμάτων σε Fortran - Mothra* [DGK+88, KO91, OC94].
- *Χρήση αναλυτικών μεθόδων* [OL94].
- *Εκτέλεση μεταλλαγμένου κώδικα* [OL94].
- *Χρήση στατιστικών μεθόδων για μείωση των μεταλλάξεων* [OP97].
- *Υπό συνθήκες έλεγχος - Constraint-based Testing Technique* [OP97].
- *«Αδύναμη» μετάλλαξη - Weak mutation* [How82, OL94].
- *Συμπαγής (ή άκαμπτη) μετάλλαξη - Firm mutation* [WH88].
- *Μετάλλαξη διαμοιρασμού ροής ελέγχου - Split–stream mutation* [KO91].
- *Μετάλλαξη με σχήματα (ομάδες – mutant schemata)* [UOH93].
- *Μετάλλαξη σε επίπεδο μεταγλωττιστή - Compiler-Integrated Program Mutation* [DKM91]

- *Τεχνικές βελτιστοποίησης μετάφρασης προγράμματος - Compiler optimization techniques* [OC94]
- *Σύστημα για προγράμματα σε Ada83* [VM98]
- *Επιλεκτική Μετάλλαξη – Selective Mutation* [VM98]
- *Μετάλλαξη αντικειμένων Java και διεπαφών επικοινωνίας - Interface mutation* [DMM01, GM01, VMBD01]
- *Μετάλλαξη κλάσεων - Class mutation* [KCM00, KCM00b]

3.1. Το σύστημα Mothra για έλεγχο προγραμμάτων σε Fortran

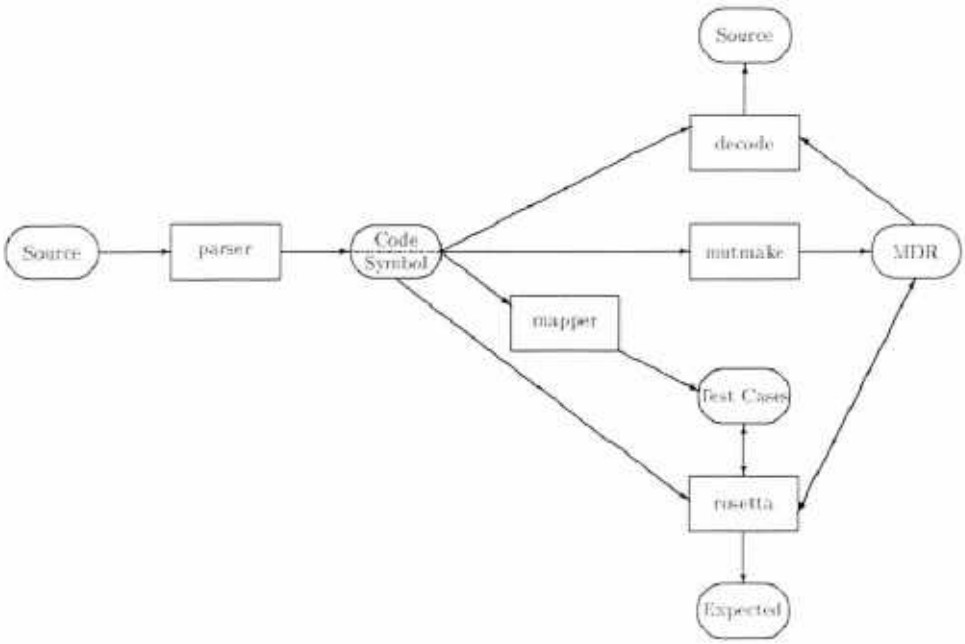
Το σύστημα Mothra είναι βασισμένο στην μέθοδο των μεταλλάξεων. Αποτελεί ένα πλήρες γλωσσικό σύστημα (language system) που μεταφράζει το υπό εξέταση πρόγραμμα σε μια ενδιάμεση μορφή κώδικα, ώστε να μπορεί να εκτελεστεί όπως και οι μεταλλάξεις του από ένα διερμηνέα (interpreter) [KO91].

Το Mothra είναι ένα σύστημα, που σε αντίθεση με αντίστοιχα προηγούμενα του, χρησιμοποιεί αυτοματοποιημένη (βασισμένη σε μαθηματικούς αλγορίθμους) αναγνώριση των ισοδύναμων μεταλλαγμένων εκδόσεων. Όπως αναφέρεται από τους δημιουργούς του [OC94] η υλοποίηση αυτού του μέρους του συστήματος, που ονομάστηκε Equalizer, βασίστηκε σε τεχνικές βελτιστοποίησης μεταγλωττιστή (compiler optimization) και επεκτείνουν τους αλγόριθμους που περιγράφηκαν αρχικά από τους Baldwin και Sayward. Οι τεχνικές αυτές αναφέρονται επιγραμματικά:

- Ανάλυση ροής δεδομένων - Data flow analysis
- Εύρεση νεκρού κώδικα - Dead code detection
- Διασπορά σταθερών - Constant Propagation
- Διασπορά μη διαφορετικότητας - Invariant Propagation
- Χρήση κοινών υπο-εκφράσεων - Common Sub-expressions
- Αναγνώριση μη διαφορετικών κύκλων - Loop Invariant Detection
- Χρήση ανύψωσης και βύθισης - Using Hoisting and Sinking

Το Mothra αποτελούν ο Godzilla, ένα υποσύστημα για τη δημιουργία δεδομένων ελέγχου (test data generator), και ο Equivalencer, που ανιχνεύει ισοδύναμες μεταλλάξεις εφαρμόζοντας την CBT μέθοδο (Constraint-Based Testing) βασιζόμενος σε στρατηγικές που χρησιμοποιούν περιοριστικές συναρτήσεις. Επίσης σημαντικό μέρος του όλου συστήματος είναι και ο διερμηνέας Rosetta. Αυτός είναι η

καρδιά του Mothra, αφού φροντίζει να κατασκευάσει και να εκτελέσει όλες τις μεταλλάξεις και το πρωτότυπο πρόγραμμα. Στο παρακάτω σχήμα δίνεται η αρχιτεκτονική του συστήματος Mothra με όλα τα υποσυστήματα που το αποτελούν [KO91].



Σχήμα 3: Αρχιτεκτονική του συστήματος Mothra

Για να ελέγξει ένα πρόγραμμα το Mothra αυτό που κάνει είναι να το περάσει από τον parser για να δημιουργήσει ένα ενδιάμεσου κώδικα αρχείο (intermediate code file) όπως και ένα αρχείο με συμβολική πληροφορία (symbol file) για τις μεταβλητές του προγράμματος και τα χαρακτηριστικά τους. Έπειτα ο ενδιάμεσος κώδικας περνά από το Mutmake υποσύστημα, ώστε να κατασκευαστούν οι εγγραφές μεταλλάξεων οι οποίες μέσα από το Rosetta θα βοηθήσουν στην κατασκευή των μεταλλάξεων. Το Rosetta με την σειρά του λαμβάνοντας τα δεδομένα από το σύνολο των test cases που έχει δημιουργήσει ο Mapper εκτελεί το γνήσιο πρόγραμμα και φυλάει τα αποτελέσματα του μαζί με τα δεδομένα που χρησιμοποίησε, για να τα εφαρμόσει και στις μεταλλάξεις.

Έπειτα το Rosetta προσαρμόζει τις εγγραφές μεταλλάξεων στον ενδιάμεσο κώδικα και παράγει εσωτερικά μια μεταλλαγμένη έκδοση του προγράμματος. Αφού την εκτελέσει ελέγχει τα αποτελέσματα με αυτά του γνήσιου προγράμματος. Αν διαφέρουν τότε η μετάλλαξη σκοτώνεται και δεν ξαναχρησιμοποιείται με άλλα δεδομένα, αλλιώς τον διατηρεί ζωντανό. Η ταχύτητα του υποσυστήματος Rosetta είναι κρίσιμη για την όλη διαδικασία, αφού οι εγγραφές μεταλλάξεων που παράγονται μπορεί να είναι πάρα πολλές.

Το σύνολο των τελεστών που εφαρμόζει το Mothra παρουσιάζεται στο παρακάτω πίνακα.

Τύπος	Περιγραφή	Κλάση
aar	Αντικατάσταση αναφοράς πίνακα με άλλη αναφορά	cca
abs	Εισαγωγή απόλυτης τιμής	pda
acr	Αντικατάσταση σταθεράς με αναφορά πίνακα	cca
aor	Αντικατάσταση αριθμητικού τελεστή	cca
asr	Αντικατάσταση μεταβλητής με αναφορά πίνακα	cca
car	Αντικατάσταση αναφοράς πίνακα με σταθερά	cca
cnr	Αντικατάσταση ονόματος πίνακα	cca
crp	Αντικατάσταση σταθεράς	pda
csr	Αντικατάσταση μεταβλητής από σταθερά	cca
der	Αλλαγή τέλους δήλωσης DO	sal
dsa	Τροποποίηση DATA δήλωσης	pda
glr	Αντικατάσταση GOTO label	sal
lcr	Αντικατάσταση λογικού τελεστή	pda
ror	Αντικατάσταση σχεσιακού τελεστή	pda
rsr	Αντικατάσταση δήλωσης RETURN	sal
san	Αντικατάσταση δηλώσεων από την TRAP	sal
sar	Αντικατάσταση αναφοράς πίνακα από μεταβλητή	cca
scr	Αντικατάσταση σταθεράς με μεταβλητή	cca
sdl	Αφαίρεση δήλωσης	sal
src	Αντικατάσταση σταθεράς κώδικα	cca
svr	Αντικατάσταση μεταβλητών	cca
uoi	Εισαγωγή unary τελεστή	pda

Πίνακας 1: Τελεστές μετάλλαξης στο Mothra

3.2. «Αδύναμη» μετάλλαξη - Weak mutation

Σύμφωνα με τον Howden [How82], που πρώτος εισήγαγε την έννοια της αδύναμης μετάλλαξης, έστω ότι P ένα πρόγραμμα και C ένα απλό συστατικό (component) του, C′ η μεταλλαγμένη έκδοση του C και P′ η έκδοση του προγράμματος που δημιουργήθηκε περιέχοντας αυτό. Η αδύναμη μετάλλαξη απαιτεί από ένα έλεγχο t να αποκαλύπτει διαφορά μεταξύ C και C′, να δίνει δηλαδή διαφορετικό αποτέλεσμα στην εκτέλεση του C′ από ότι αυτή του C. Αυτό στην περίπτωση της ισχυρής μετάλλαξης δεν θα ήταν αρκετό, αφού εκεί ελέγχεται το τελικό αποτέλεσμα από την συνολική εκτέλεση των P και P′. Για αυτό και η συγκεκριμένη προσέγγιση ονομάζεται «αδύναμη», αφού χρειάζεται ένα λιγότερο αυστηρό, πιο αδύναμο, σύνολο δεδομένων ελέγχου από ότι η ισχυρή εκδοχή της μεθόδου των μεταλλάξεων.

Σύμφωνα πάλι με τον Howden στο συγκεκριμένο είδος μετάλλαξης τα είδη των συστατικών είναι πέντε, χωρίς όμως να έχει οριστεί από τον ίδιο σαφώς η έννοια του συστατικού:

1. Αναφορά μεταβλητής (Variable reference)
2. Ανάθεση μεταβλητής (Variable assignment)
3. Αριθμητική έκφραση (Arithmetic expression)
4. Σχεσιακή έκφραση (Relational expression)
5. Δισήμαντη έκφραση (Boolean expression)

Ο Howden πρότεινε ότι δεν είναι ανάγκη να επανεκτελείται ο κώδικας για κάθε μετάλλαξη, αλλά όλες οι μεταλλάξεις για ένα συστατικό (program component) θα μπορούσαν να ελεγχθούν με ένα μοναδικό έλεγχο.

Σύμφωνα με τους Offutt και Lee [OL94] ο ορισμός της ισχυρής μετάλλαξης εξαρτάται σημαντικά από τους τελεστές μετάλλαξης (mutation operators) και με τον τρόπο υλοποίησης τους ανάλογα με την γλώσσα. Αντίστοιχα ο ορισμός στην περίπτωση της «αδύναμης» προσέγγισης σχετίζεται άμεσα με την έννοια του συστατικού του προγράμματος (program component), και αυτό με την σειρά του από την γλώσσα προγραμματισμού.

Προηγούμενες μελέτες στην «αδύναμη» μετάλλαξη αναφέρουν ως πρώτη απόπειρα το σύστημα του Hamlet (το 1977) [Ham77]. Αυτό ήταν ένα σύστημα ελέγχου ενσωματωμένο (embedded) σε μεταγλωττιστή (compiler), το οποίο αν και αρκετά διαφορετικό από τα μετέπειτα συστήματα εφάρμοζε την «αδύναμη»

μετάλλαξη. Επίσης μελέτες όπως αυτή του Marick έδωσαν αποτελέσματα [Mar91, Mar93] που υποστηρίζουν την υπόθεση ότι η αδύναμη μετάλλαξη έχει σχεδόν την ίδια αποτελεσματικότητα με την ισχυρή. Οι Horgan και Mathur αναλύοντας την μέθοδο [HM90] κατέληξαν ότι υπό συνθήκες τα δεδομένα ελέγχου που δημιουργούνται με την αδύναμη μετάλλαξη αναμένεται να ικανοποιήσουν και τις συνθήκες της ισχυρής με αυξημένη πιθανότητα να παρέχουν υψηλή αξιοπιστία. Επίσης ο έλεγχος υπό συνθήκες (constraint-based testing) [DO91] υποστηρίζει έμμεσα την αδύναμη μετάλλαξη. Ίσως να εννοείται ότι περιορίζοντας τα δεδομένα ελέγχου ελαττώνεται το εύρος των σεναρίων ελέγχου του προγράμματος. Τότε το υποσύνολο αυτών που εξετάζονται με την αδύναμη μέθοδο πλησιάζει το αντίστοιχο σύνολο της ισχυρής μετάλλαξης.

3.3. Συνδυασμός με ανάλυση ροής δεδομένων

Οι Girgis και Woodward [GW85] (το 1985) υλοποίησαν ένα σύστημα ελέγχου προγραμμάτων σε Fortran-77 που συνδύαζε αδύναμη μετάλλαξη και ανάλυση ροής δεδομένων (data flow analysis). Αυτό το σύστημα δεν εκτελούσε τις μεταλλάξεις, αλλά προσπαθούσε με αναλυτικές μεθόδους να βρει αν έπρεπε να τις «σκοτώσει» ή όχι. Το συγκεκριμένο έχει δύο βασικά μειονεκτήματα, παρόλο που σαν προσέγγιση είναι οικονομικότερη υπολογιστικά από ότι αντίστοιχες που βασίζονται σε εκτέλεση των μεταλλαγμένων εκδόσεων ενός προγράμματος. Μπορεί να αποφασίσει για την εξόντωση περιορισμένου είδους μεταλλάξεων, ενώ από την άλλη απαιτεί περιορισμένου εύρους δομικά στοιχεία της γλώσσας για τα οποία τελεί τους ελέγχους. Αυτό κρίθηκε αρνητικά από τους Offutt και Lee στην αξιολόγηση της «αδύναμης» μετάλλαξης [OL94], διότι απέκλειε αρκετά αποτελεσματικά στοιχεία, χωρίς όμως να διευκρινίζουν ποια.

3.4. Η συμπαγής μετάλλαξη - firm mutation

Οι Woodward και Halewood εισήγαγαν την έννοια του firm mutation (συμπαγής μετάλλαξη) [WH88] προτείνοντας ότι δεν είναι πάντα απαραίτητο ο έλεγχος να γίνεται είτε στο τέλος του όλου προγράμματος είτε μετά από κάθε δομικό στοιχείο του (program component). Σύμφωνα με αυτή την έννοια η αδύναμη μετάλλαξη είναι τεχνική τοπικού εύρους, ενώ η ισχυρή καθολικού. Κάθε μεταλλαγμένο πρόγραμμα θα μπορούσε να θεωρηθεί μη ισοδύναμο ελέγχοντας τα

αποτελέσματα του σε οποιοδήποτε εύρος από το τοπικό μέχρι το καθολικό, καθορίζοντας το σημείο ανοχής ενός λάθους και το πέρασμα του οποίου θα αποτελεί ικανή συνθήκη για την εξόντωση της μετάλλαξης. Είναι παρόμοια ιδέα με αυτή του Morell, ο οποίος είχε αναφερθεί στη διάρκεια ενός λάθους και την σημασία που έχει αυτό στον έλεγχο προγραμμάτων με την μέθοδο της εμφύτευσης λαθών (fault-based testing) [Mor90]. Ο τελευταίος μάλιστα αναφέρεται σε αρκετούς λόγους για τους οποίους μπορεί ένα λάθος να μην προκαλέσει διαφορά στο τελικό αποτέλεσμα του προγράμματος, όπως συγκάλυψη τιμών (masking of data values) και διόρθωση λαθών από το ίδιο το πρόγραμμα. Αυτό δικαιολογεί και την αφερεγγυότητα ελέγχων με δεδομένα (test cases), που ικανοποιούν τους κανόνες της προσβασιμότητας (reachability) και αναγκαιότητας (necessity) κατά τους DeMillo και Offutt [DO91, OP97], αλλά δεν είναι απαραίτητα ικανοί για να ικανοποιήσουν τον κανόνα της επάρκειας (sufficiency).

3.5. Το σύστημα LEONARDO – Μια τροποποίηση του Mothra

Οι Offutt και Lee τροποποίησαν το σύστημα Mothra, για να εφαρμόσουν «αδύναμη» μετάλλαξη, δημιουργώντας το LEONARDO [OL94] (Looking at Expected Output Not After Return but During Operation – Έλεγχος Αναμενόμενου Αποτελέσματος Όχι Μετά την Επιστροφή αλλά Κατά την Διεργασία). Το συγκεκριμένο σύστημα εφαρμόζει έλεγχο των μεταλλάξεων εκτελώντας τις (execution-based mutation testing). Χρησιμοποιεί συνολικά 22 τελεστές (operators) [DGK+88, KO91] για να ελέγξει τα προγράμματα, όσους δηλαδή και το Mothra [DGK+88], σε αντίθεση με το σύστημα των Girgis και Woodward που αξιοποιεί μόνο τρεις από αυτούς. Ελέγχει καταστάσεις του πρωτότυπου και των μεταλλαγμένων προγραμμάτων σε ενδιάμεσα σημεία κατά την εκτέλεση τους. Συγκεκριμένα αυτό που κάνει είναι να εκτελεί το πρωτότυπο πρόγραμμα μέχρι το σημείο που πρέπει να ελεγχθεί και σώζει την κατάσταση του. Έπειτα εκτελεί τις μεταλλαγμένες εκδόσεις που σχετίζονται με εκείνο το σημείο ελέγχοντας την αντίστοιχη κατάσταση. Έτσι εξοικονομείται χρόνος από την επανεκτέλεση κώδικα για κοινό σημείο μετάλλαξης, αφού το πρωτότυπο πρόγραμμα εκτελείται μόνο μια φορά για μια ομάδα μεταλλαγμένων εκδόσεων του.

Οι δημιουργοί του υποστηρίζουν ότι υλοποιήθηκε με τρόπο που θα μπορούσε να θεωρηθεί ότι ικανοποιεί την προσέγγιση της συμπαγούς ή άκαμπτης μετάλλαξης

(firm mutation). Αυτό φαίνεται να είναι εφικτό χάρη στην αρχιτεκτονική του συστήματος. Υπάρχει ένα σύστημα-πράκτορας (agent) που παρακολουθεί την εξέλιξη της εκτέλεσης του μεταλλαγμένου κώδικα και αλλάζει την ροή του δυναμικά ανάλογα με τα αποτελέσματα από την εκτέλεση κάθε κόμβου. Προφανώς λοιπόν αφού το σύστημα διαχειρίζεται την ροή εκτέλεσης του προγράμματος θα μπορεί να εφαρμόσει και συμπαγή μετάλλαξη σχετικά εύκολα.

3.6. *Μετάλλαξη με διαμοιρασμό ροής ελέγχου (Split-stream mutation)*

Υπάρχει επίσης και η εκδοχή που περιγράφηκε από τους King και Offutt σε σχετική εργασία τους χωρίς όμως να έχει υλοποιηθεί (όταν γράφτηκε η εργασία όπου αναφέρεται) και η οποία αποκαλείται split-stream (σε ελεύθερη μετάφραση «διαμοιρασμού ροής ελέγχου») [KO91]. Αυτή είναι πιο κοντά στα λεγόμενα του Howden που είχε υποστηρίξει ότι δεν είναι αναγκαία η επανεκτέλεση κώδικα για τον έλεγχο κάθε μετάλλαξης ενός συστατικού προγράμματος (program component). Είτε αναλυτική είτε βασισμένη στην εκτέλεση των μεταλλαγμένων εκδόσεων (όπως την περιέγραψαν οι King και Offutt ως split-stream execution) αυτή η εκδοχή φυλάσσει την κατάσταση του προγράμματος στο σημείο πριν την εκτέλεση του στοιχείου του οποίου είναι επιθυμητός ο έλεγχος και έπειτα εξετάζει όλες τις δυνατές περιπτώσεις μετάλλαξης ξεχωριστά. Σαν προσέγγιση βρίσκεται πιο κοντά σε αυτή που προτείνεται από την παρούσα εργασία, για αυτό και θα αναλυθεί κάπως πιο διεξοδικά παρακάτω

3.7. Χρήση σχημάτων μετάλλαξης – mutant schemata

Οι Untch, Offutt και Harrold πρότειναν την εφαρμογή σχημάτων μετάλλαξης (ομάδες – mutant schemata) [UOH93] σαν λύση στις χαμηλές ταχύτητες ελέγχου που έδιναν οι μέχρι τότε μέθοδοι, εκτιμώντας βελτίωση ως και 300%. Την παρουσίασαν ως μια μέθοδο που δημιουργεί ένα μετα-πρόγραμμα που περιέχει όλες τις μεταλλάξεις και έπειτα μεταφράζεται και εκτελείται. Είναι μια μέθοδος που μπορεί να υλοποιηθεί ευκολότερα από άλλες που βασίζονται σε διερμηνευτικά (interpretive) συστήματα, όπως το σύστημα Mothra, είναι μεταφέρσιμη και ανεξάρτητη πλατφόρμας λογισμικού ή υλικού, ενώ χρησιμοποιεί τις ίδιες παραμέτρους (μεταφραστή –compiler- και υποστήριξη σε χρόνο εκτέλεσης – run-time support) με αυτές κατά την ανάπτυξη του υπό εξέταση προγράμματος.

Ένα σχήμα (schema) είναι ένα είδος πρότυπου, που αποτελείται από μια μετά-μετάλλαξη (metamutant) και μια μετά-συνάρτηση (metaprocedure). Για να γίνει κατανοητή όμως η μέθοδος ας θεωρήσουμε τον κανόνα AOR (Arithmetic Operator Replacement), ο οποίος αναφέρει ότι κάθε εμφάνιση ενός αριθμητικού τελεστή πρέπει να αντικαθιστάται από καθένα από τους υπόλοιπους δυνατούς αντίστοιχους τελεστές και από κάποιους ειδικούς επιπλέον. Έστω το παρακάτω πρόγραμμα για τον υπολογισμό της τετραγωνικής ρίζας με την μέθοδο του Νεύτωνα, όπως αυτός υπάρχει και στην εργασία των Untch, Offutt και Harrold.

```
1   PROCEDURE Newton(Number:REAL; VAR Sqrt:REAL);
2   (* Find square root using Newton's method. *)
3   VAR
4       NewGuess, Delta, Epsilon : REAL;
5   BEGIN
6       Epsilon := 0.001;
7       NewGuess := (Number / 2.0) + 1.0;
8       Sqrt := 0.0;
9       Delta := NewGuess - Sqrt;
10      WHILE Delta > Epsilon DO
11          Sqrt := NewGuess;
12          NewGuess := (Sqrt+(Number/Sqrt))/2.0;
13          Delta := NewGuess - Sqrt;
14          IF Delta < 0.0 THEN
15              Delta := -Delta;
16          END;
17      END; (* END WHILE *)
18  END Newton;
```

Σχήμα 4: Συνάρτηση Newton (υπολογίζει τη ρίζα αριθμού)

Στην γραμμή 9 υπάρχει ο πρωτότυπος κώδικας

Delta := NewGuess – Sqrt;

για τον οποίο μπορούν να υπάρξουν σύμφωνα με τον κανόνα AOR έξι δυνατές μεταλλάξεις, οι οποίες φαίνονται παρακάτω:

1) Delta := NewGuess + Sqrt;
2) Delta := NewGuess * Sqrt;

3) Delta := NewGuess / Sqrt;
4) Delta := NewGuess **MOD** Sqrt;
5) Delta := NewGuess **.LEFTOP.** Sqrt;
6) Delta := NewGuess **.RIGHTOP.** Sqrt;

Όλες οι παραπάνω μεταλλάξεις μπορούν να αναπαρασταθούν με ένα αφηρημένο αριθμητικό τελεστή τον ArithOp, όπως φαίνεται παρακάτω:

Delta := NewGuess ***ArithOp*** Sqrt;

ή αν είχαμε μια μετά-συνάρτηση την Aorr, στην οποία θα είχαμε συμπεριλάβει όλες τις δυνατές περιπτώσεις μετάλλαξης που προαναφέρθηκαν, θα μπορούσε να αλλάξει ο κώδικας στην γραμμή 9 όπως φαίνεται παρακάτω (όπου το 62 είναι ο κωδικός που αντιστοιχεί σε συγκεκριμένο τελεστή):

Delta := ***Aorr***(NewGuess, Sqrt, 62);

Η υλοποίηση της συνάρτησης Aorr [UOH93] είναι απλή, ανήκει στην πρώτη κατηγορία των μετα-τελεστών (metaoperators), αλλά περιέχει όλες τις μεταλλάξεις. Οι παράμετροι που δέχεται είναι δύο πραγματικοί αριθμοί που αποτελούν τους συντελεστές της αριθμητικής πράξης (στο παράδειγμα μας οι NewGuess και Sqrt) και ένας ακέραιος που είναι το διακριτικό του τελεστή που θα χρησιμοποιηθεί. Όσο για τις metaoperand συναρτήσεις δεν υπάρχει παράδειγμα στην συγκεκριμένη εργασία.

```
PROCEDURE AOrr
  (LeftOp,RightOp:REAL; ChangePt:INTEGER):REAL;
BEGIN
    CASE  Variant(ChangePt)  OF
        aoADD:    RETURN  LeftOp + RightOp;
      | aoSUB:    RETURN  LeftOp - RightOp;
      | aoMULT:   RETURN  LeftOp * RightOp;
      | aoDIV:    RETURN  LeftOp / RightOp;
      | aoMOD:    RETURN  LeftOp MOD RightOp;
      | aoRIGHT:  RETURN  RightOp;
      | aoLEFT:   RETURN  LeftOp;
    ELSE
        Error( "AOrr CASE out of range" );
        RETURN 0.0;
    END;
END AOrr;
```

Σχήμα 5: Τελεστές μετάλλαξης σε σχηματική μορφή (mutant schema)

Αυτό που είναι σημαντικό από αυτή την προσέγγιση είναι ότι το πρόγραμμα P δημιουργεί ένα meta-mutant πρόγραμμα M, για το οποίο γίνεται μετάφραση και σύνδεση (compilation & linking) μια μόνο φορά και έπειτα μπορεί δυναμικά να εκτελείται αλλάζοντας κάποιες παραμέτρους, που δέχονται οι μετά-συναρτήσεις ανάλογα με το ποια μετάλλαξη έχει σειρά να εκτελεστεί. Όταν δημιουργείται η μετά-μετάλλαξη ενός προγράμματος P δημιουργείται και μια λίστα D με τα διακριτικά για κάθε επιλογή μετάλλαξης. Ένας κοινός για όλους τους ελέγχους οδηγός (driver) αρχικοποιεί κάθε φορά την κατάλληλη μετάλλαξη που πρόκειται να εκτελεστεί και μεριμνεί για διάφορα διαχειριστικά θέματα, όπως χειρισμό των περιπτώσεων ελέγχου (test cases), των δεδομένων εισόδου-εξόδου, των εξαιρέσεων, συγκρίνει τα αποτελέσματα της κάθε μετάλλαξης με τα αντίστοιχα του πρωτότυπου προγράμματος και καταγράφει συμπεράσματα. Επίσης υπολογίζει το βαθμό μετάλλαξης MS καθώς η διαδικασία βρίσκεται σε εξέλιξη.

Οι συγγραφείς διακρίνουν σε δύο κατηγορίες αυτές τις συναρτήσεις, στους μετά-τελεστές (metaoperators) και στις συναρτήσεις metaoperand. Η διαφορά τους είναι ότι οι πρώτες συναρτήσεις είναι στατικές, δημιουργούνται δηλαδή μια φορά μόνο και υπάρχουν σαν βιβλιοθήκη για να εφαρμόζονται σε όλα τα προς έλεγχο προγράμματα. Αντιθέτως οι συναρτήσεις της δεύτερης κατηγορίας δημιουργούνται δυναμικά ανάλογα με το πρόγραμμα. Κατά τα λοιπά όλες εξαρτώνται από τα

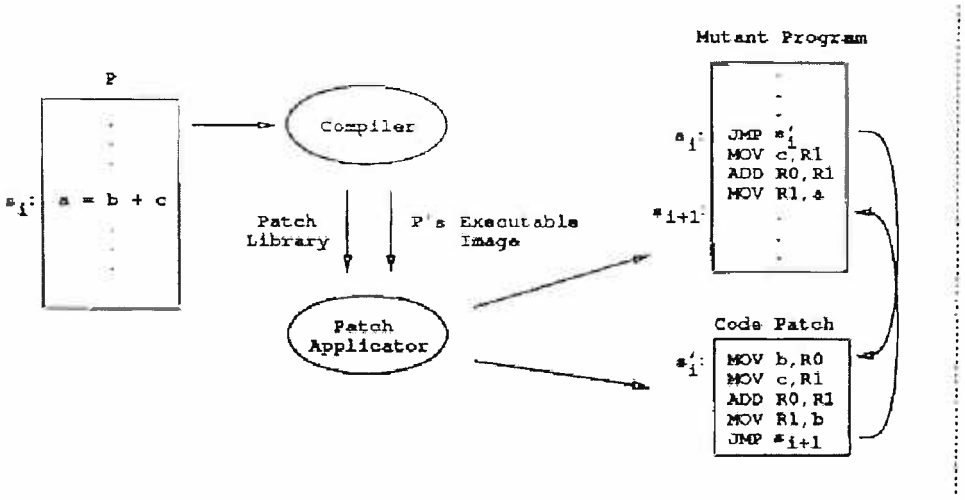
χαρακτηριστικά της γλώσσας στην οποία είναι γραμμένο το υπό εξέταση πρόγραμμα, αποτελούν ομάδες μεταλλάξεων από τις οποίες ορίζεται σε χρόνο εκτέλεσης ποια θα εφαρμοστεί.

3.8. Compiler-Integrated Program Mutation

Σύμφωνα με τους DeMillo, Krauser και Mathur οι παραδοσιακοί μεταφραστές (compilers), μέχρι το 1991 που έγραφαν την συγκεκριμένη εργασία ("Compiler-Integrated Program Mutation", [DKM91]), εφάρμοζαν έλεγχο σχετικά με την κάλυψη δηλώσεων (statement) και κλάδων (branch). Κάτι τέτοιο φαίνεται να γίνεται στον μεταγλωττιστή της JAVA (έκδοση 1.4), αφού για παράδειγμα δεν επιτρέπει να τεθεί ως αληθής (true) η συνθήκη σε κλήση while δήλωσης, γιατί οδηγεί σε αόριστο κύκλο (infinite), αν και δεν βρέθηκε να αναφέρεται στις προδιαγραφές της γλώσσας. Κατά την διάρκεια όμως της μεταγλώττισης υπάρχει διαθέσιμη πληροφορία ικανή για να εφαρμοστούν πιο εξελιγμένες τεχνικές ελέγχου. Χρησιμοποιώντας αυτή την πληροφορία λοιπόν πρότειναν μια τεχνική για την δημιουργία μεταλλαγμένων εκδόσεων ενός προγράμματος. Διαπίστωσαν ότι η μετάφραση καθεμιάς μετάλλαξης ως ξεχωριστό πρόγραμμα είναι μια πλεονάζουσα προσέγγιση, αφού κάθε «νέο» πρόγραμμα διαφέρει ελάχιστα από το πρωτότυπο. Για αυτό πρότειναν κατά την μετάφραση του πρωτότυπου προγράμματος P να δημιουργείται το εκτελέσιμο P_{exec} και ένα σύνολο από μπαλώματα του (patches) P_{pat}. Κάθε φορά που πρέπει να ελεγχθεί μια μετάλλαξη προστίθεται στο πρόγραμμα το αντίστοιχο μπάλωμα. Έτσι δημιουργείται η απαραίτητη μεταλλαγμένη έκδοση του πρωτότυπου προγράμματος έτοιμη προς εκτέλεση σε μορφή εκτελέσιμου κώδικα. Αυτό συνεπάγεται οικονομία στο χρόνο που απαιτείται για την μεταγλώττιση του κώδικα (source code) σε εκτελέσιμο (executable code), το οποίο έχει μεγάλη αξία λόγω του μεγάλου πλήθους των μεταλλάξεων που δημιουργούνται για κάθε πρόγραμμα.

Η προσθήκη ενός «μπαλώματος» είναι μια διαδικασία με αυξημένη αξιοπιστία και χαμηλό κόστος. Η διαδικασία είναι απλή και στηρίζεται στην δομή ενός προγράμματος γραμμένο σε γλώσσα μηχανής. Το μόνο που χρειάζεται είναι να προστεθεί στο τέλος του προγράμματος ο μεταλλαγμένος κώδικας (που θα είναι επίσης σε γλώσσα μηχανής). Έπειτα στο σημείο που είναι επιθυμητή η μετάλλαξη να δημιουργηθεί ένα «άλμα» (jump) προς εκείνη την διεύθυνση μνήμης, ενώ προστίθεται και μια πινακίδα (label) στην εντολή αμέσως μετά. Αυτό είναι το σημείο

που επιστρέφει η εκτέλεση του προγράμματος τελειώνοντας το block με την μετάλλαξη. Η συγκεκριμένη διαδικασία παρουσιάζεται στο παρακάτω σχήμα.



Σχήμα 6: Εφαρμογή τελεστή μετάλλαξης με μορφή μπαλώματος

Βέβαια όλα τα παραπάνω είναι προσδοκίες των συγγραφέων την περίοδο που έγραψαν την σχετική εργασία (πριν το 1991). Όπως πίστευαν, αυτή η προσέγγιση θα ήταν σε θέση να παρέχει αισθητή επιτάχυνση της διαδικασίας και μείωση του κόστους εκτέλεσης του ελέγχου προγραμμάτων με την μέθοδο των μεταλλάξεων. Η υλοποίηση του αναγκαίου συστήματος που θα εφάρμοζε την τεχνική όπως την περιέγραψαν το υλοποιούσαν τότε και δεν αναφέρουν πειραματικά αποτελέσματα αξιολόγησης της, ενώ δεν βρέθηκαν νεότερες σχετικές αναφορές. Το σημαντικό υπέρ της προσέγγισης αυτής, σύμφωνα πάντα με τους εμπνευστές της, είναι ότι το πρόγραμμα που ελέγχεται παραμένει αρκετά όμοιο με το πρωτότυπο όσο αφορά την συμπεριφορά του (operational behaviour).

Αυτό που δεν αναφέρεται πουθενά στην προσέγγιση των DeMillo, Krauser και Mathur είναι το τι γίνεται με την εκτέλεση του κώδικα πριν και μετά το σημείο μετάλλαξης. Φαίνεται ότι ακολουθούν την προσέγγιση της εκτέλεσης ολόκληρης της μετάλλαξης μέχρι τέλους. Αυτό σημαίνει την επανεκτέλεση κοινών μερών των προγραμμάτων και κατά συνέπεια «άσκοπη» καθυστέρηση της όλης διαδικασίας.

3.9. Σύστημα μετάλλαξης προγραμμάτων σε Ada83

Το συγκεκριμένο σύστημα αναπτύχθηκε από τον Offutt σε συνεργασία με την εταιρεία Reliable Software Technologies το 1993 στα πλαίσια προσπάθειας επιδοτούμενης από την NASA. Προσπάθησε να απαντήσει σε κρίσιμα ερωτήματα για να αναπτυχθεί μια θεωρητική βάση για την μετάλλαξη προγραμμάτων σε Ada. Είναι ερωτήσεις, που (σύμφωνα με τους δημιουργούς του συγκεκριμένου συστήματος) θα έπρεπε να λαμβάνονται υπόψη σε κάθε προσπάθεια για δημιουργία αντίστοιχων συστημάτων για καινούριες γλώσσες όπως η JAVA [VM98]. Συγκεκριμένα:

1. Πως είναι δυνατό να ελεγχθούν προγράμματα που χρησιμοποιούν ισχυρούς τύπους (strong typing) και απόκρυψη πληροφορίας (information-hiding mechanisms);
2. Πως είναι δυνατό να ελεγχθούν μηχανισμοί μνήμης όπως δυναμικές δομές δεδομένων (dynamic data structures) και στατικές δομές αποθήκευσης (static data storage structures);
3. Πως μπορούν να ελεγχθούν σύνθετοι μηχανισμοί ελέγχου όπως η διαχείριση εξαιρέσεων (exception handling) και η ανάθεση εργασιών (tasking);
4. Πως η χρήση ταυτόχρονης επεξεργασίας (concurrent processing) επηρεάζει την εφαρμογή του ελέγχου με μεταλλάξεις;

Το συγκεκριμένο σύστημα μετάλλαξης βασίζεται σε μεταγλώττιση του κώδικα και προσπαθεί να συμπεριλάβει όλες τις πιθανές μεταλλάξεις σε ένα εκτελέσιμο. Αυτό το κάνει γιατί όπως υποστηρίζεται στο [VM98] ο εκτελέσιμος κώδικας που παράγεται από μεταγλωττιστή είναι πιο αξιόπιστος και έχει καλύτερες επιδόσεις από τον αντίστοιχο ενός διερμηνέα. Το σύστημα πρότεινε πέντε κατηγορίες τελεστών μετάλλαξης για αντικατάσταση τελεσταίων (operands), για δηλώσεις και εκφράσεις, όπως και για κάλυψη και ανάθεση εργασιών προγράμματος. Σημαντικές βέβαια είναι και οι οδηγίες που προέκυψαν για μελλοντικά συστήματα.

3.10. Επιλεκτική μετάλλαξη – Selective Mutation

Συχνά ο έλεγχος με την μέθοδο των μεταλλάξεων παράγει αρκετές μεταλλάξεις και για αυτό απαιτεί αρκετό χρόνο. Η επιλεκτική μετάλλαξη είναι μια ιδέα του Offutt και βασίστηκε σε δύο παρατηρήσεις:

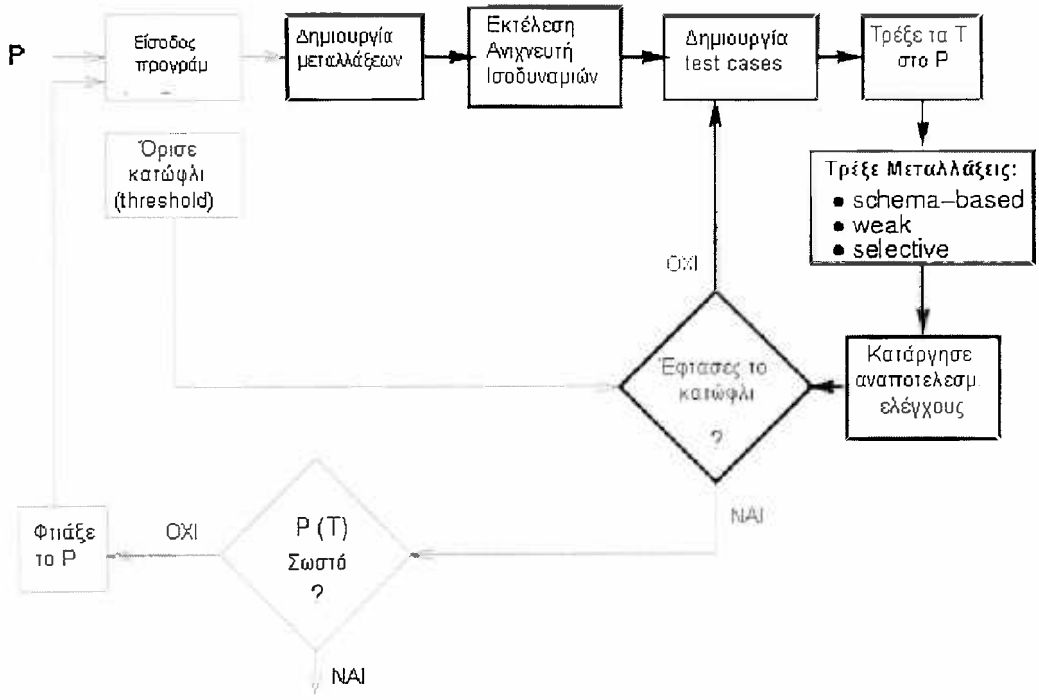
- Ανεπαρκής υλοποίηση της μεθόδου σχετικά με τις επιδόσεις στο Mothra.
- Πλεονασμός περιπτώσεων που οδήγησε σε μη αποδεκτές επιδόσεις.

Η προσέγγιση αυτή επιλέγει προσεγγιστικά ορισμένους τελεστές που είναι στην ουσία διαφορετικοί από τους άλλους και ελέγχει μόνο αυτούς [ORZ93, VM98, MB99]. Έτσι δημιουργείται μικρότερο πλήθος μεταλλάξεων, από ότι στην πλήρη τεχνική.

Στους έξι από τους 22 τελεστές μετάλλαξης, που υπάρχουν στο Mothra, αντιστοιχεί συνήθως το 40-60 % των εκδόσεων, που πρόκειται να ελεγχθούν. Οι τελεστές, που χρησιμοποιούνται για Fortran-77, είναι εισαγωγής απόλυτης τιμής, αντικατάστασης αριθμητικών, λογικών, σχεσιακών και unary τελεστών. Στην περίπτωση της Ada83 προτείνονται κάποιοι τελεστές ακόμα για δημιουργία εξαιρέσεων σε περίπτωση μηδενικών δεδομένων, υπερχείλισης ή «υποχείλισης» (underflow) και αντικατάστασης καλούμενου υποπρογράμματος. Επίσης πρέπει να δοκιμάζονται και κάποιοι ειδικοί τελεστές εξαρτώμενοι από την γλώσσα. Συγκεκριμένα στην περίπτωση επιλεκτικής μετάλλαξης για προγράμματα σε Ada83 πρέπει να δοκιμάζονται τελεστές κάλυψης πολλαπλών συνθηκών, μετάλλαξης στην ανάθεση εργασιών προγράμματος (tasking) και ελέγχου μέλους (membership test replacement).

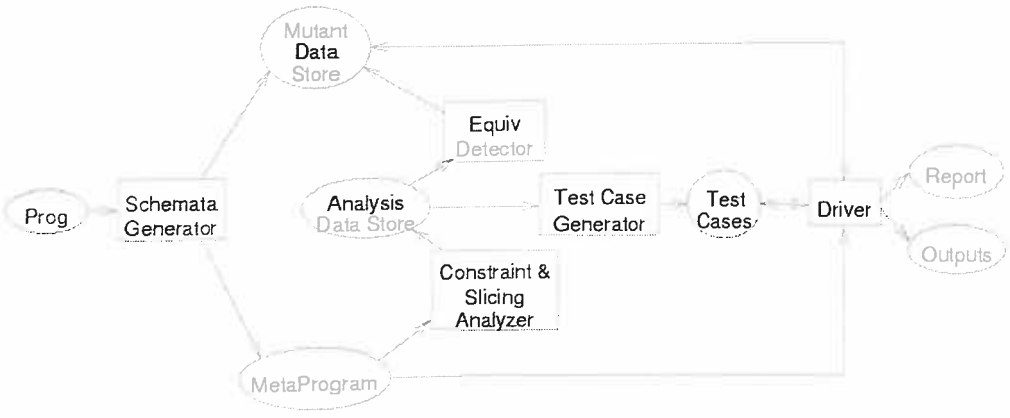
3.11. Η πιο πρόσφατη προσέγγιση

Σε πρόσφατη εργασία τους όμως οι Offutt και Untch [OU00] (βλ. επίσης [Off95]) πρότειναν μια καινούρια προσέγγιση της διαδικασίας, που κατορθώνει την αυτοματοποίηση της σε μεγάλο βαθμό. Στο σχήμα τα αυτοματοποιημένα βήματα είναι τα κουτιά σε έντονο πλαίσιο. Είναι σημαντικό να σημειωθεί όμως ότι ακόμα και σε αυτή την εκδοχή είναι προφανές ότι αφήνεται ο έλεγχος των αποτελεσμάτων από την εκτέλεση των πρωτότυπων ή μεταλλαγμένων προγραμμάτων στο χειριστή του συστήματος. Αυτό είναι πιθανό να εξακολουθεί να επιβαρύνει σημαντικά την όλη διαδικασία, δεδομένου και του πλήθους των μεταλλάξεων.



Σχήμα 7: Νέος τρόπος εκτέλεσης ελέγχου μεταλλάξεων

Η αρχιτεκτονική του συστήματος που υλοποιεί την παραπάνω διαδικασία φαίνεται στο παρακάτω σχήμα και στην ουσία περιλαμβάνει όλα τα επιμέρους συστήματα, που έχουν αναπτύξει οι ερευνητικές ομάδες των Offutt και Untch τα τελευταία δεκαπέντε χρόνια.



Σχήμα 8: Αρχιτεκτονική αποδοτικού συστήματος ελέγχου μεταλλάξεων

3.12. Μετάλλαξη αντικειμένων Java και διεπαφών

Σύμφωνα με τους Bieman, Ghosh και Alexander [BGA01] ο έλεγχος της κατάστασης ενός αντικειμένου δεν μπορεί να εφαρμοστεί στατικά, γιατί η ίδια η κατάσταση εξαρτάται από την εκτέλεση και την ροή του προγράμματος, από την ακολουθία των μεθόδων που καλούνται και όχι από τις τιμές των παραμέτρων τους μόνο. Αυτό τους οδήγησε να ορίσουν τελεστές μετάλλαξης για διεπαφές επικοινωνίας (Interfaces) που υλοποιούνται από ένα μεγάλος εύρος κλάσεων αντί να ασχοληθούν με κάθε κλάση χωριστά. Η γενικότερη ιδέα είναι να δημιουργούνται λάθη που σχετίζονται μόνο με το μέρος της επικοινωνίας και ολοκλήρωσης των επιμέρους συστατικών ενός συστήματος. Αυτό ονομάστηκε μετάλλαξη διεπαφών (interface mutation). Στο επίπεδο της ολοκλήρωσης συστημάτων εμφανίστηκε τα τελευταία χρόνια κάτι αντίστοιχο ως μετάλλαξη ολοκλήρωσης (integration mutation) [DMM01, GM01, VMBD01]. Πέρα λοιπόν από τους κοινούς (default), όπως τους ονομάζουν, τελεστές μετάλλαξης όρισαν και για:

- τους container τύπους των διεπαφών (interfaces) Collection και List που βρίσκονται στο πακέτο (package) java.util.
- τους Iterator τύπους της αντίστοιχης διεπαφής στο πακέτο java.util.
- και για την αφηρημένη κλάση (abstract class) InputStream του πακέτου java.io.

Βέβαια από τους κοινούς τελεστές μετάλλαξης αναφέρουν μόνο τρεις στην συγκεκριμένη εργασία (αύξηση ή μείωση κατά μια μονάδα και ανάθεση σταθερής τιμής). Αυτό ίσως να σημαίνει ότι δεν συμπεριέλαβαν τους υπόλοιπους.

Η συνολική ιδέα που είχαν ήταν να υπάρχει μια ObjectMutationEngine η οποία θα παρακολουθούσε την ώρα εκτέλεσης το πρόγραμμα και θα εκτελούσε τις απαραίτητες μεταλλάξεις και ελέγχους. Αυτό για να γίνει χρησιμοποίησαν ένα σύστημα (code instrumenter) που αναλύοντας την δομή του προγράμματος προσέθετε ανάλογα κλήσεις συναρτήσεων μετάλλαξης ή μη από βιβλιοθήκες σε συγκεκριμένους κόμβους. Όταν το πρόγραμμα κατά την εκτέλεση του έβρισκε μια κλήση σε μετάλλαξης της παραπάνω μηχανής για κάποιο αντικείμενο το αναζητούσε από μια λίστα καταχωρημένων (registered). Για να καταχωρηθεί ένα αντικείμενο μετάλλαξης προστίθεται στον κώδικα μια κατάλληλη κλήση από την μηχανή. Αυτή η υλοποίηση σύμφωνα με τους σχεδιαστές της είναι αρκετά ευέλικτη (flexible), γιατί μπορεί να

εμφυτεύσει μεγάλη ποικιλία μεταλλάξεων σε προγράμματα, ενώ έχει και την δυνατότητα για εύκολη επέκταση και υποστήριξη νέων μοντέλων.

Οι καινούριοι τελεστές μετάλλαξης που προτάθηκαν από την συγκεκριμένη προσέγγιση είναι:

- Χρήση αναφοράς σε αντίγραφο της αναφερόμενης μεταβλητής
- Αρχικοποίηση περίπτωσης (instance) αντικείμενου συμβατού με τον τύπο του αναφερόμενου
- Άδειασμα μιας συλλογής (Collection) ή λίστας (List)
- Αφαίρεση στοιχείου από συλλογή/ λίστα
- Πρόσθεση στοιχείου σε συλλογή/ λίστα
- Μετάλλαξη των στοιχείων μιας συλλογής/ λίστας
- Αλλαγή σειράς μέρους στοιχείων μιας λίστας
- Παράληψη στοιχείου μιας περίπτωσης Iterator (iterate-επαναλαμβάνω)
- Παράληψη n bytes από την είσοδο (InputStream)

3.13. Μετάλλαξη κλάσεων - Class Mutation

Οι Kim, Clark και McDermid προσπάθησαν να προσαρμόσουν το σύστημα Mothra σύμφωνα με την αντικειμενοστρεφή τεχνική μετάλλαξης τους, που ονομάζουν "Class Mutation" [KCM00, KCM00b]. Για τις δοκιμές τους τροποποίησαν το σύστημα Mothra, σύμφωνα με την γραμματική της Java, ώστε να μπορέσουν να εφαρμόσουν μετάλλαξη σε αντίστοιχα προγράμματα. Η τεχνική που προτείνουν θυμίζει ισχυρή μετάλλαξη τροποποιημένη, ώστε να λαμβάνει υπόψη τα χαρακτηριστικά των αντικειμενοστρεφών γλωσσών. Σύμφωνα με τους συγγραφείς υπάρχουν πολλά είδη λαθών που εμφανίζονται μόνο σε προγράμματα γραμμένα σε αυτής της κατηγορίας γλώσσες, τα οποία οφείλονται στα ιδιαίτερα χαρακτηριστικά των τελευταίων. Για να τα καλύψουν πρότειναν τους παρακάτω επιπλέον τελεστές, στα πλαίσια της τεχνικής Class Mutation (μετάλλαξης κλάσεων):

- Αντικατάσταση τύπου κλάσης από άλλο συμβατό τύπο
- Αλλαγή μιας δημιουργίας *instance* με κάποιο άλλο ίδιου ή συμβατού τύπου
- Αλλαγή σειράς παραμέτρων κάποιας μεθόδου στην δήλωση της
- Αφαίρεση δήλωσης κάποιας υπερφορτωμένης (*overloaded*) μεθόδου
- Αλλαγή σειράς παραμέτρων μεθόδου κατά την κλήση της
- Σταδιακή μείωση παραμέτρων κλήσης μεθόδου

- Αφαίρεση δήλωσης πεδίου όταν υπάρχει στην *super* κλάση
- Πρόσθεση δήλωσης πεδίου ίδιου ονόματος στην *super* κλάση
- Αφαίρεση δήλωσης μιας *overriding* μεθόδου
- Αντικατάσταση προσδιορισμού πρόσβασης (*access modifier*) από άλλους
- Πρόσθεση ή αφαίρεση του προσδιορισμού *static*
- Αφαίρεση διαχείρισης εξαίρεσης
- Αλλαγή διαχείρισης εξαίρεσης με κλήση παραγωγής της και ανάποδα
- Αφαίρεση τοπικής μεταβλητής με ίδιο όνομα με κάποιο πεδίο
- Αντιμετάθεση δηλώσεων μεταξύ τοπικών μεταβλητών και πεδίων με ίδιο όνομα

Προσπαθώντας να αξιολογήσουν την τεχνική που πρότειναν και για την δημιουργία των απαραίτητων δεδομένων επέλεξαν τρεις μεθόδους αντικειμενοστρεφούς ελέγχου. Αρχικά χρησιμοποίησαν την προσέγγιση των Doong και Frankl [DF94], όπου εξετάζει την περίπτωση μια συγκεκριμένη αλληλουχία βημάτων να οδηγήσει το αντικείμενο της κλάσης που ελέγχεται στην «σωστή» αφηρημένη κατάσταση ("correct" abstract state). «Ένα αντικείμενο O1 είναι ισοδύναμο μέσω παρατήρησης με το αντικείμενο O2 της κλάσης C (observationally equivalent) αν και μόνο αν είναι αδύνατη η διάκριση τους κατά την εκτέλεση των διεργασιών της κλάσης και σχετιζόμενων με αυτή κλάσεων». Αυτού του τύπου η ισοδυναμία εκφράζει τις έννοιες της ενθυλάκωσης (encapsulation) και της απόκρυψης πληροφορίας από μη συγγενικές οντότητες. Η δεύτερη μέθοδος ήταν αυτή των Kirani και Tsai [KT94]. Αυτοί πρότειναν την εξέταση της ακολουθίας των μεθόδων που εκτελούνται από μια κλάση ("method sequence specification"), ώστε να εντοπιστεί η σωστή ή μη συμπεριφορά του αντικειμένου. Τέλος χρησιμοποιήθηκε η προσέγγιση του Kung [KSG+94], η οποία επικεντρώνει τον έλεγχο στην κατάσταση του αντικειμένου (Object State Testing) και λαμβάνει υπόψη τα πεδία τα οποία χρησιμοποιούνται σε συνθήκες απόφασης κρίσιμες για την ροή του προγράμματος.

Καμιά όμως από τις παραπάνω τεχνικές ελέγχου αντικειμενοστρεφών προγραμμάτων δεν κατάφερε να ξεχωρίσει, ενώ δεν φάνηκε να είναι και ιδιαίτερα αποτελεσματικές παρόλο που διατυπώνουν την ειδικότητα τους για έλεγχου αυτής της κατηγορίας προγραμμάτων. Καμία επίσης δεν χρησιμοποιεί κάλυψη δηλώσεων (statement coverage), αν και ορισμένες περιπτώσεις θα έπρεπε να καλύπτονται σύμφωνα με τους υποστηρικτές της Class Mutation προσέγγισης. Τέλος φάνηκε ότι

είναι μια αρκετά απαιτητική και δαπανηρή εργασία η εύρεση δεδομένων που να αποκαλύπτουν τις υπολειπόμενες διαφορές, ώστε τελικά να σκοτώσουν τις ζωντανές μεταλλάξεις.

3.14. Τι προσέφερε (πλεονεκτήματα) σε σχέση με προηγούμενες προσεγγίσεις;

Η μέθοδος μεταλλάξεων βασίστηκε στις δύο παραδοχές που προαναφέρθηκαν, στην υπόθεση ικανότητας και αποτελεσματικότητας του προγραμματιστή και στην επίδραση σύζευξης λαθών. Αυτές ήταν ικανές για να την κάνουν αντικαταστάτη της γενικότερης μεθόδου εμφύτευσης λαθών στο πρόγραμμα που δημιουργούσε αρκετά μεγαλύτερο πλήθος «λανθασμένων» εκδόσεων του. Έτσι ο έλεγχος με την μέθοδο εμφύτευσης λαθών αρκέστηκε στην χρήση της μεθόδου μετάλλαξης του προγράμματος έχοντας τα ίδια αποτελέσματα με μικρότερο κυρίως υπολογιστικό κόστος. Παρόλα αυτά όμως ακόμα και με αυτή την ελάττωση του πλήθους των εκδόσεων που δημιουργούνται προς έλεγχο, η μέθοδος εξακολουθεί να είναι αρκετά δαπανηρή και αναζητούνται τρόποι περαιτέρω βελτίωσης της.

3.15. Υπάρχοντα προβλήματα (μειονεκτήματα) προς επίλυση

Στον ορισμό της μεθόδου τονίζεται η εμφύτευση **ενός και μόνο** λάθους σε κάθε μονάδα προγράμματος. Αυτός τελικά είναι και ο λόγος για τον οποίο η συγκεκριμένη τεχνική είναι αρκετά δαπανηρή υπολογιστικά [DGK+88, OL94, How82]. Αυτή η προσέγγιση έχει αυξημένο κόστος και χαμηλή ταχύτητα εκτέλεσης, κάτι που οφείλεται στις περισσότερες υλοποιήσεις στους παρακάτω παράγοντες:

1. Μεγάλος αριθμός μεταλλάξεων
2. Κόστος μεταγλώττισης κάθε μετάλλαξης
3. Κόστος εκτέλεσης κάθε μετάλλαξης.
4. Μη αυτοματοποιημένη εισαγωγή δεδομένων ελέγχου
5. Μη αυτοματοποιημένος έλεγχος αποτελεσμάτων
6. Μη αυτοματοποιημένη εύρεση ισοδύναμων μεταλλάξεων

Αναλυτικότερα ο μεγάλος αριθμός των μεταλλαγμένων εκδόσεων που δημιουργούνται ακόμα και για μικρά προγράμματα είναι κρίσιμος παράγοντας καθυστέρησης της μεθόδου. Σημαντικό ζητούμενο μιας νέας τεχνικής που θα εφαρμόζει την μέθοδο θα πρέπει να είναι η μείωση των μεταλλάξεων για κάθε

τελεστή (operator) της γλώσσας στην οποία είναι γραμμένος ο κώδικας. Προς αυτή την κατεύθυνση κινούνται προσεγγίσεις όπως η χρήση στατιστικών μεθόδων για μείωση των μεταλλάξεων [OP97], η χρήση «σχημάτων μετάλλαξης» (mutation schemas) [UOH93] ή ακόμα και τεχνικές βελτιστοποίησης μεταγλωττιστή (compiler optimization techniques) [OC94].

Επιπλέον η δημιουργία αρκετών εκδόσεων του προγράμματος που περιέχουν ένα λάθος, επιβαρύνει (σημαντικά) την χρονική διάρκεια της όλης διαδικασίας λόγω της μεταγλώττισης και της εκτέλεσης τους [OP97]. Ακόμα και προσεγγίσεις με χρήση ενδιάμεσου κώδικα όπως είναι το Mothra ή την ανακατεύθυνση του προγράμματος με χρήση «μπαλωμάτων» χρειάζονται ένα σημαντικό χρόνο για την μεταγλώττιση και σύνδεση (linking) όλων αυτών των μικρών ενδιάμεσων μερών. Πρέπει λοιπόν να βρεθεί μια προσέγγιση, ώστε να μειωθεί ο χρόνος μεταγλώττισης και εκτέλεσης όλων των μεταλλάξεων.

Ακόμα τα δεδομένα ελέγχου εισάγονται από τον χρήστη-ελεγκτή όπως και τα αποτελέσματα εξόδου για κάθε σενάριο ελέγχονται από αυτόν, ώστε να διαπιστωθεί η εγκυρότητας τους ή όχι [OP97]. Είναι αναγκαίο λοιπόν να βρεθεί τρόπος αυτοματοποιημένου ελέγχου για αυτά τα βήματα της μεθόδου.

3.16. Το πρόβλημα εύρεσης ισοδύναμης μετάλλαξης

Το πρόβλημα εύρεσης της ισοδυναμίας μεταξύ του πρωτότυπου προγράμματος και της μετάλλαξης που ελέγχεται αναφέρεται σε όλες τις προσεγγίσεις. Στην πλειοψηφία των περιπτώσεων παρουσιάζεται ότι γίνεται από τον άνθρωπο, αφού δεν φαίνεται σαν πρόβλημα να μπορεί να έχει μια εύκολα υλοποιήσιμη αυτοματοποιημένη λύση. Όλες οι μερικώς αυτοματοποιημένες προσεγγίσεις [OC94, OP97] το καλύτερο που προσπαθούν να επιτύχουν είναι να μειώσουν το πλήθος αυτών που καταλήγουν στην κρίση του χειριστή του συστήματος.

Οι Offutt και Pan [σελ. 173, OP97] υποστηρίζουν ότι το πρόβλημα εύρεσης ισοδύναμων μεταλλάξεων είναι μέρος του προβλήματος εύρεσης πραγματοποιήσιμων μονοπατιών. Αντίστοιχα μέρη του ίδιου γενικότερου προβλήματος είναι η εύρεση των μη προσπελάσιμων δηλώσεων ενός προγράμματος (unreachable statements) και των απραγματοποίητων DU μονοπατιών (infeasible DU-paths). Αυτό σημαίνει ότι

αλλαγές σε σημεία του κώδικα, τα οποία δεν εκτελούνται, συνεπάγονται ισοδύναμες εκδόσεις.

Στην εργασία των Offutt και Pan [OP97] αναφέρονται τεχνικές εύρεσης ισοδύναμων μεταλλάξεων που βασίζονται σε μαθηματικά ορισμένες περιοριστικές συναρτήσεις (constraint-based functions), όπως είναι:

- Άρνηση – Negation
- Διαμοιρασμός περιορισμού – Constraint splitting
- Σύγκριση σταθερών τιμών – Constant comparison

Οι Offutt και Craft [OC94] αναφέρουν ότι η εφαρμογή τεχνικών διαμοιρασμού προγράμματος (program slicing) ίσως να είναι χρήσιμη για την βελτίωση της αποτελεσματικότητας της εύρεσης ισοδύναμων μεταλλάξεων. Μάλιστα αυτό μελετάται και από τους Hierons, Harman και Danicic σε πιο πρόσφατη εργασία [HHD99]. Ενώ προτείνει μελλοντικά την μελέτη του παράλληλου προγραμματισμού και κατά πόσο αυτός μπορεί να βοηθήσει προς βελτίωση της ανίχνευσης ισοδυναμίας στην ανάλυση κύκλων (loops), ειδικά σε αναδρομικές περιπτώσεις όπου μια μεταβλητή ορίζεται σε συνάρτηση του εαυτού της ενώ υπάρχει και χρήση της τιμής της. Επίσης προτείνουν για ανακάλυψη των ισοδυναμιών μεταξύ διαφορετικών εκδόσεων προγραμμάτων την χρήση τεχνικών βελτιστοποίησης κώδικα στην φάση της μεταγλώττισης, ανάλυση ροής δεδομένων και την κατανομή των ισοδυναμιών ανάλογα με τον τύπο του τελεστή μετάλλαξης.

Κεφάλαιο 4ο: Η γλώσσα JAVA

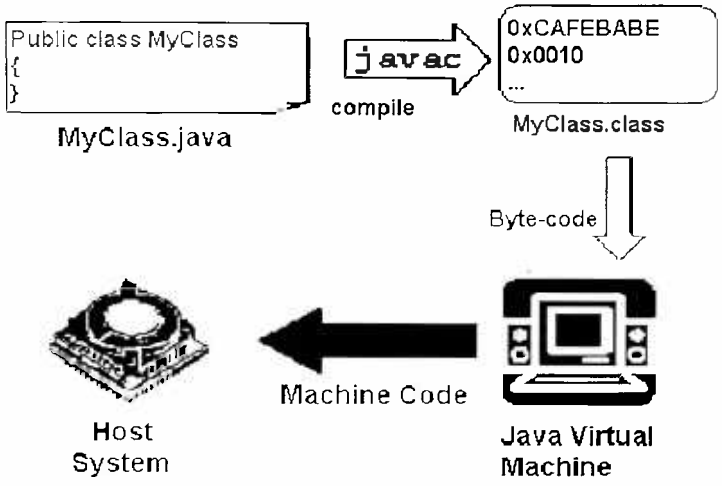
4.1. Χαρακτηριστικά της JAVA

Η JAVA σχεδιάστηκε για να ανταποκριθεί στις προκλήσεις ανάπτυξης εφαρμογών στα πλαίσια ετερογενών και δικτυακά κατανεμημένων συστημάτων. Κύρια πρόκληση, μεταξύ άλλων, είναι η ασφαλής μεταφορά των εφαρμογών, που να καταναλώνουν το ελάχιστο των πόρων του συστήματος, να μπορούν να τρέχουν σε οποιαδήποτε πλατφόρμα υλικού και λογισμικού (hardware/ software) και να μπορούν να επεκταθούν δυναμικά. Η JAVA δημιουργήθηκε ως μέρος ερευνητικού έργου για την ανάπτυξη προχωρημένου λογισμικού για μία μεγάλη ποικιλία δικτυακών συσκευών και εμπεριεχομένων (embedded) συστημάτων. Σκοπός ήταν η ανάπτυξη

μιας μικρής αξιόπιστης, μεταφέρσιμης, κατανεμημένης σε πραγματικό χρόνο (real – time) πλατφόρμας λειτουργίας.

Είναι μία αντικειμενοστρεφής γλώσσα προγραμματισμού, η οποία παρέχει μία «καθαρή» και αποδοτική object – based αναπτυξιακή πλατφόρμα. Παράλληλα, μεταξύ των κύριων χαρακτηριστικών της είναι και η απλότητά. Οι θεμελιώδεις έννοιες της γίνονται γρήγορα κατανοητές και οι προγραμματιστές μπορούν να είναι παραγωγικοί από τα πρώτα βήματα. Το γεγονός, επίσης ότι η JAVA μοιάζει αρκετά με τη C++, την καθιστά αρκετά οικία, ενώ ταυτόχρονα αφαιρεί όλες τις λεπτομέρειες που δημιουργούν πρόβλημα σε πιο άπειρους προγραμματιστές. Η γλώσσα αυτή έχει σχεδιαστεί για τη δημιουργία «υψηλής αξιοπιστίας» λογισμικό. Παρέχει εκτεταμένο έλεγχο κατά το χρόνο της μεταγλώττισης (compile – time checking), ακολουθούμενο από ένα δεύτερο επίπεδο ελέγχου κατά το χρόνο του τρεξίματος (run – time checking). Το μοντέλο της διαχείρισης μνήμης (memory management model) είναι εξαιρετικά απλό (όλα τα αντικείμενα δημιουργούνται με τον τελεστή *new*). Δεν υπάρχουν άμεσα ορισμένοι δείκτες (pointer), τύποι δεδομένων (data types) και η απελευθέρωση μνήμης γίνεται αυτόματα (garbage collection).

Για να εξυπηρετήσει την ποικιλία από πλατφόρμες υλικού και όλα τα πιθανά λειτουργικά περιβάλλοντα, ο μεταφραστής (compiler) της JAVA παράγει bytecodes – μια αρχιτεκτονικά ουδέτερη ενδιάμεση μορφή κώδικα. Η διερμηνευμένη (interpreted) αυτή μορφή του κώδικα της JAVA σχεδιάστηκε για να επιτύχει αποδοτική μεταφερσιμότητα. Επίσης η γλώσσα προσδιορίζει με ακρίβεια το μέγεθος των βασικών τύπων δεδομένων (data types) και τη συμπεριφορά των αριθμητικών τελεστών της. Τα προγράμματα είναι ίδια σε κάθε πλατφόρμα και δεν υπάρχουν ασυνέπειες τύπων δεδομένων σε διαφορετικές αρχιτεκτονικές. Η πλατφόρμα της αρχιτεκτονικής ουδετερότητας και μεταφερσιμότητας στην JAVA είναι γνωστή ως *JAVA Virtual Machine*(**JVM**). Είναι ο προσδιορισμός μιας «αφηρημένης» (abstract) μηχανής για την οποία οι μεταφραστές της JAVA μπορούν να παράγουν κώδικα. Η όλη διαδικασία παραγωγής εκτελέσιμου κώδικα από τον κώδικα σε JAVA φαίνεται στο σχήμα 8.



Σχήμα 9: Μετατροπή αρχείου java σε εκτελέσιμο κώδικα

Η πλατφόρμα της JAVA επιτυγχάνει υψηλή απόδοση με την υιοθέτηση ενός σχήματος μέσω του οποίου ο διερμηνέας μπορεί να τρέχει στην μέγιστη ταχύτητα χωρίς να υπάρχει η ανάγκη ελέγχου του run – time περιβάλλοντος. Η αυτόματη συγκομιδή της αχρησιμοποίητης μνήμης (garbage collector) τρέχει ως χαμηλής προτεραιότητας διαδικασία (background thread), επιβεβαιώνοντας υψηλή πιθανότητα ότι η μνήμη είναι διαθέσιμη όταν αυτό απαιτείται και βελτιστοποιώντας την απόδοση του συστήματος.

Ενώ ο μεταφραστής της JAVA είναι περιορισμένος στο στατικό έλεγχο του χρόνου μεταγλώττισης, η γλώσσα και το run-time σύστημα είναι δυναμικά (dynamic) στα στάδια της σύνδεσης. Οι κλάσεις συνδέονται όπως χρειάζεται, ενώ η φάση σύνδεσης (link phase) ενός προγράμματος είναι απλή, προοδευτική (incremental) και συντηρητική (lightweight). Όταν υπάρχει δηλωμένη κλάση που χρησιμοποιείται στο πρόγραμμα, τότε λαμβάνεται υπόψη μόνο αυτή και όχι όλες οι κλάσεις που υπάρχουν στο φάκελο που περιλαμβάνεται με την εντολή import. Σε αντίθεση με τις C/C++, που περιλαμβάνουν κομμάτια από τα '.h' αρχεία και τον κώδικα των αρχείων που συμπεριλαμβάνονται (#include), στην JAVA η import δεν αντιγράφει κώδικα στο πρόγραμμα που εκτελείται, απλά δηλώνει που μπορεί να βρεθούν οι αναγκαίες χρησιμοποιούμενες κλάσεις κατά την εκτέλεση του. Καινούρια μέρη (modules) κώδικα μπορεί να συνδεθούν, όταν απαιτηθεί από μία ποικιλία πηγών, ακόμη και από πηγές που προέρχονται από ένα δίκτυο.

Συγκεντρωτικά, τα χαρακτηριστικά της JAVA είναι τα ακόλουθα:

- Αντικειμενοστρεφής (object – oriented),απλή (simple).
- Σταθερή (robust) και ασφαλής (secure).
- Αρχιτεκτονικά ουδέτερη (architecture neutral) και μεταφέρσιμη (portable).
- Υψηλή απόδοση (high performance).
- Χρησιμοποιεί διερμηνέα (Interpreted), βασίζεται σε διαφορετικές λογικές ενότητες εκτέλεσης (threaded) και είναι δυναμική (dynamic).

4.2. Αυστηρός έλεγχος κατά το χρόνο μεταγλώττισης και κατά το χρόνο τρεξίματος

Ο JAVA compiler εκτελεί εκτεταμένο και αυστηρό έλεγχο κατά το χρόνο μεταγλώττισης, ώστε λάθη συντακτικά να ανακαλυφθούν γρήγορα, προτού να αναπτυχθεί το πρόγραμμα. Πολλοί από τους ελέγχους κατά το χρόνο μεταγλώττισης μεταφέρονται στους run – time ελέγχους, για να ελεγχθεί η συνέπεια και να εξασφαλιστεί ευελιξία.

Η γλώσσα [VMSpec] είναι strongly typed, το οποίο σημαίνει ότι κάθε μεταβλητή και κάθε έκφραση έχουν τύπο γνωστό κατά την μεταγλώττιση του προγράμματος. Οι τύποι διακρίνονται σε πρωτογενείς (primitive), που είναι ορισμένοι από την γλώσσα και τους έχει ανατεθεί μια λέξη κλειδί, και αναφορικούς (reference), ενώ υπάρχει και ένας ειδικός τύπος ο null (μηδενικός). Δεν υποστηρίζονται απαριθμημένοι (enumerated) τύποι με την έννοια που αυτοί συναντώνται στη C ή σε άλλες γλώσσες.

4.2.1 Πρωτογενείς τύποι δεδομένων

Η γλώσσα παρέχει τελεστές ακέραιων τιμών, όπως αριθμητικής σύγκρισης, αριθμητικής πράξης, βαθμωτής αύξησης και μείωσης τιμής, μετάπτωσης (casting), λογικούς και μετατόπισης (shift) τελεστές σε επίπεδο bit. Για πραγματικούς αριθμούς παρέχει τελεστές για αριθμητικές συγκρίσεις και πράξεις, για βαθμωτή αύξηση και μείωση και μετάπτωση. Όσο για τις λογικές τιμές υπάρχουν σχεσιακοί και λογικοί τελεστές, ενώ σε δηλώσεις ελέγχου ροής προγράμματος πρέπει να χρησιμοποιούνται μόνο λογικές εκφράσεις. Η γλώσσα υποστηρίζει την διάκριση θετικού και αρνητικού μηδέν. Μεταξύ τους είναι αληθής η σύγκριση (0.0 == -0.0) ενώ είναι ψευδής η (0.0 > -0.0). Η JAVA φροντίζει να κλείνει κύκλο τις δυνατές ακέραιες τιμές οπότε αποφεύγεται έτσι η υπερχείλιση (overflow). Στην περίπτωση των πραγματικών

αριθμών σε περίπτωση υπερχείλισης επιστρέφεται προσημασμένο άπειρο, σε περίπτωση «υποχείλισης» (underflow) επιστρέφεται προσημασμένο μηδέν και αν δεν ορίζεται η πράξη τότε επιστρέφεται NaN. Δεν μπορεί να συμβεί μετάπτωση από αριθμητικές τιμές σε δισήμαντες (boolean).

Μόνο η διαίρεση και ο τελεστής υπολοίπου ακεραίων προκαλούν εξαίρεση (ArithmeticException) σε περίπτωση που ο δεύτερος τελεσταίος είναι μηδέν. Σαν αποτέλεσμα μη παραδεκτών πράξεων, όπως διαίρεση του μηδέν με μηδέν, η JVM επιστρέφει την τιμή NaN (Not-a-Number). Αν γίνει σύγκριση του NaN με κάποιο τελεστή από τους <, <=, >, και >= το αποτέλεσμα είναι ψευδές (false).

4.2.2 Αναφορικοί τύποι δεδομένων

Υπάρχουν τρεις κατηγορίες αναφορικών τύπων: σε περιπτώσεις κλάσεων, σε διεπαφές (interfaces) και σε πίνακες. Δεν υπάρχει η έννοια του δείκτη όπως αυτός ορίζεται σε γλώσσες όπως την C++ και αυτό για χάρη της απλότητας της γλώσσας. Οι τιμές αυτών των τύπων όμως μοιάζουν σα να είναι δείκτες στα αντίστοιχα αντικείμενα ή σε μηδενική αναφορά (null reference) σε περίπτωση που δεν αντιστοιχεί σε κάποιο συγκεκριμένο.

4.2.3 Μεταβλητές στην JAVA

Υπάρχουν εφτά είδη μεταβλητών στην JAVA. Υπάρχουν μεταβλητές κλάσης (class) αλλά και περίστασης (instance), στοιχεία πίνακα, παράμετροι μεθόδου, παράμετροι κατασκευαστικής μεθόδου (constructor), διαχείρισης εξαίρεσης (exception-handler parameter variable) και τοπικές μεταβλητές (local variables). Όλες οι αναθέσεις σε μεταβλητές ελέγχονται την στιγμή της μεταγλώττισης του προγράμματος ως προς την συμβατότητα ανάθεσης (assignment compatibility). Τα αντικείμενα πρέπει πάντα να ανήκουν σε συγκεκριμένη κλάση, όπως και οι μεταβλητές να έχουν καθορισμένο τύπο. Βέβαια μια έκφραση ή μια μεταβλητή που έχει τύπο interface, μπορεί να αναφερθεί σε οποιοδήποτε αντικείμενο του οποίου η κλάση υλοποιεί το συγκεκριμένο interface.

Σημαντικό είναι να αναφερθεί ότι παρόλο που είναι δυνατό να οριστεί σε οποιοδήποτε σημείο της μεθόδου μια μεταβλητή, δεν επιτρέπεται από τον μεταγλωττιστή η χρήση της σε κάποια συνθήκη αν δεν έχει πρώτα αρχικοποιηθεί. Επίσης δεν επιτρέπεται να επιστρέφεται η τιμή της αν δεν έχει αρχικοποιηθεί στο ίδιο

επίπεδο εκτέλεσης ανεξάρτητα από το αν έχει γίνει κάτι τέτοιο εντός κάποιου εσωτερικού loop (ανεξάρτητα από τη συνθήκη εκτέλεσης του).

4.2.4 Διαχείριση εξαιρέσεων

Όταν το πρόγραμμα παραβιάζει τους κανόνες της JAVA τότε η JVM δημιουργεί μια «εξαίρεση» για να δηλώσει το συγκεκριμένο λάθος. Το ίδιο το πρόγραμμα πάλι μπορεί να προκαλέσει αντίστοιχη εξαίρεση με την εντολή *throw*. Κάθε εξαίρεση ανήκει στην κλάση *Throwable* ή σε κάποια υποκλάση της. Η διαχείριση των εξαιρέσεων γίνεται με την χρήση εντολών *catch* που ακολουθούν δηλώσεις *try*. Ο μηχανισμός διαχείρισης εξαιρέσεων συνεργάζεται με το μοντέλο συγχρονισμού της γλώσσας, ώστε οι συγχρονισμένες διεργασίες να ολοκληρώνουν την δουλειά τους, πριν σταματήσει το πρόγραμμα λόγω έλλειψης διαχειριστή για την εξαίρεση που προέκυψε στην διάρκεια εκτέλεσης της. Η αποτελεσματική διαχείριση των εξαιρέσεων από το ίδιο το πρόγραμμα είναι κρίσιμος παράγοντας για την σταθερότητα του και την ασταμάτητη λειτουργία του.

Εξαιρέσεις προκαλούνται όταν εκτελείται μια μη αποδεκτή κατάσταση, όπως αναφορά σε στοιχείο πίνακα με α/α μεγαλύτερο του μεγέθους του, όταν παρατηρείται λάθος στο φόρτωμα ή την σύνδεση προγράμματος ή όταν χρησιμοποιείται μνήμη πέρα από αυτή που είναι διαθέσιμη. Άλλος λόγος είναι να προκληθεί εξαίρεση μετά από κλήση εντολής throw από το ίδιο το πρόγραμμα.. Τέλος μπορεί να προκληθεί εξαίρεση αν σταματήσει η λογική ενότητα (thread) που εκτελείται ή αν απλά δημιουργηθεί πρόβλημα στην ίδια την JVM.

4.3. Δηλώσεις ελέγχου ροής εκτέλεσης (Conditional statements)

Οι δηλώσεις ελέγχου ροής εκτέλεσης, επαναληπτικές ή μη, που υποστηρίζονται από την JAVA παρουσιάζονται παρακάτω, χωρίς περαιτέρω εξήγηση καθότι η σύνταξη και λειτουργία τους θεωρούνται γνωστές:

- if...else
- switch - case
- while
- do while
- for

4.4. Υπερφόρτωση συναρτήσεων

Σε μια οντοκεντρική (αντικειμενοστρεφή) γλώσσα όπως η JAVA είναι δυνατό να γίνει υπερφόρτωση (overloading) των συναρτήσεων, ώστε να είναι δυνατή η κλήση τους με ποικίλους συνδυασμούς παραμέτρων. Για παράδειγμα έστω ότι υπήρχε σε μια κλάση μια μέθοδος που λαμβάνει ως παραμέτρους δύο ακέραιους για να τους προσθέσει. Θα μπορούσε να υπάρχει με το ίδιο όνομα στην ίδια κλάση μια άλλη μέθοδος που θα δεχόταν άλλες παραμέτρους (διαφορετική σειρά τύπων ή και διαφορετικό πλήθος), όπως δύο πραγματικούς αριθμούς για να κάνει αντίστοιχη δουλειά. Αυτό διευκολύνει τον προγραμματιστή που θα χρησιμοποιήσει την συγκεκριμένη κλάση στο να χρησιμοποιεί τις μεθόδους με σιγουριά χωρίς να χάνει δεδομένα από αναγκαστικές μεταπτώσεις σε χρόνο εκτέλεσης του κώδικα (run-time casting).

4.5. Dynamic Loading και Binding

Η μεταφέρσιμη και διερμηνευμένη φύση της JAVA παράγει ένα δυναμικό και δυναμικά επεκτάσιμο σύστημα. Οι κλάσεις μπορεί να συνδεθούν όπως χρειάζεται και να γίνουν download μέσα από δίκτυα. Ο κώδικας επαληθεύεται πριν να περάσει στον διερμηνέα για εκτέλεση.

Η JAVA λύνει το «μόνιμο πρόβλημα του recompilation» -οποτεδήποτε προστίθενται μία καινούρια μέθοδος ή μία καινούρια instance variable σε μία κλάση, χρειάζονται recompilation όλες οι κλάσεις που αναφέρονται σε αυτή-. Ο compiler δεν μεταγλωττίζει τα references σε αριθμητικές τιμές – αντ' αυτού, περνά συμβολική αναφορική πληροφορία στον byte code verifier και στον διερμηνέα (interpreter). Ο τελευταίος εκτελεί ανάλυση (resolution) του τελικού ονόματος μία φορά, όταν οι κλάσεις συνδέονται. Όταν το όνομα αναλύεται, η αναφορά γράφεται ως αύξων αριθμητικό (offset), επιτρέποντας στον διερμηνέα να τρέχει με την μέγιστη ταχύτητα.

Τελικά, το layout της αποθήκευσης των objects δεν αποφασίζεται από τον compiler. Το layout των objects γίνεται κατά τον χρόνο τρεξίματος και αποφασίζεται από τον interpreter. Updated κλάσεις με καινούριες instance variables ή μεθόδους μπορεί να συνδεθούν χωρίς να επηρεάσουν τον υπάρχοντα κώδικα.

Οι κλάσεις στην JAVA έχουν μία run – time αναπαράσταση. Υπάρχει μία κλάση *Class*. Αν κάποιος χειρίζεται ένα αντικείμενο, μπορεί να βρει σε ποια κλάση ανήκει, ανεξάρτητα από το αν ξέρει τον τύπο του αντικειμένου ή όχι. Είναι, επίσης,

δυνατή η αναζήτηση του ορισμού μιας κλάσης δοθέντος αλφαριθμητικού (string) με το όνομά της. Αυτό σημαίνει ότι μπορεί να πληκτρολογήσει κανείς το όνομα του τύπου δεδομένων και να το έχει δυναμικά συνδεδεμένο (dynamically linked) στο τρέχον σύστημα.

4.6. Single Inheritance και Interfaces

Η JAVA υποστηρίζει απλή κληρονομικότητα (single inheritance), που σημαίνει ότι κάθε κλάση στοιχεία μπορεί να κληρονομεί στοιχεία μόνο από μία υπέρ-κλάση (superclass). Όμως, αυτή η απλότητα – παρά το γεγονός ότι εξασφαλίζει την ευκολότερη χρήση, υλοποίηση και «λιγότερη» πολυπλοκότητα της γλώσσας – στην ιεραρχία κλάσεων είναι περιοριστική, ειδκά όταν υπάρχει μία συμπεριφορά που χρειάζεται να χρησιμοποιηθεί από κλάσεις που βρίσκονται σε διαφορετικά κλαδιά του ίδιου δέντρου.

Η JAVA έχει μία άλλη ιεραρχία ξεχωριστά από την ιεραρχία της main class. Πρόκειται για μία ιεραρχία κλάσεων μικτής συμπεριφοράς. Κατόπιν, όταν δημιουργείται μία καινούρια κλάση, έχει μόνο μία superclass, αλλά μπορεί να διαλέξει διαφορετική συμπεριφορά από την άλλη ιεραρχία. Αυτή η άλλη ιεραρχία είναι η interface ιεραρχία. Ένα JAVA interface είναι μία συλλογή από «αφαιρετικές συμπεριφορές», που μπορούν να αναμιχθούν σε μία κλάση και να της προσθέσουν μία συμπεριφορά που δεν υποστηρίζεται από της superclasses της τρέχουσας. Πιο συγκεκριμένα, ένα JAVA interface αποτελείται από abstract ορισμούς μεθόδων και σταθερών, χωρίς να περιλαμβάνει instance variables και υλοποίησης μεθόδων.

4.7. Modularity στη JAVA

Η JAVA υποστηρίζει το χωρισμό ενός μεγάλου προγράμματος σε modules. Πιο συγκεκριμένα, ένα application πρόγραμμα σε JAVA αποτελείται από έναν αριθμό κλάσεων. Αν το πρόγραμμα είναι σωστά σχεδιασμένο, αυτές οι κλάσεις αντανακλούν ενθυλάκωση (encapsulation) και απόκρυψη πληροφορίας (information hiding). Είναι μία αντικειμενοστρεφής γλώσσα προγραμματισμού και δεν υποστηρίζει ξεχωριστές συναρτήσεις (functions) και διαδικασίες (procedures) παρά μόνο ως μέλη κάποιας κλάσης.

Κεφάλαιο 5ο: Βελτιωμένη προσέγγιση

Λαμβάνοντας υπόψη την διαθέσιμη αρθρογραφία σχετικά με το θέμα της εφαρμογής της μεθόδου των μεταλλάξεων στον έλεγχο προγραμμάτων και προσπαθώντας να το προσεγγίσουμε με σκοπό την μείωση του κόστους της όλης διαδικασίας προέκυψε ένα σημαντικό δίλημμα. Από την μια υπάρχει ένα σύστημα που εφαρμόζει την μέθοδο σε γλώσσα Fortran σε αρκετά ικανοποιητικά αποδεκτά επίπεδα κόστους. Αυτό το σύστημα έχει την αρχή του (σαν έργο) πριν δεκαπέντε χρόνια και σαφώς φανερώνει την δυσκολία που υπάρχει στο να δημιουργηθεί ένα αντίστοιχο σύστημα, που να είναι λειτουργικά αποδεκτό και να έχει ανάλογη αποτελεσματικότητα. Αυτό όμως δεν σημαίνει ότι δεν υπάρχει η πιθανότητα ο αρχικός σχεδιασμός να έγινε με βάση τα δεδομένα της εποχής και της γλώσσας της οποίας προγράμματα θα ελέγχονταν και πάνω σε αυτόν να συνεχιζόταν η έρευνα με διαρκώς ανανεώσιμες ελπίδες. Επίσης, δεν σημαίνει ότι δεν πρέπει να δοκιμαστεί όποια άλλη προσέγγιση αν υπάρχει κάποια υποψία με βάση την σχετική θεωρία και αρθρογραφία ότι μπορεί να προσφέρει λύσεις σε άλυτα προβλήματα της προηγούμενης. Πόσο μάλλον όταν ακόμα φαίνεται ότι σημαντικό κόστος της προσέγγισης του Offutt έγκειται στον μη αυτοματοποιημένο έλεγχο των αποτελεσμάτων από την εκτέλεση κάθε έκδοσης του προγράμματος (πρωτότυπης ή μεταλλαγμένης).

Επίσης υπήρχε το δίλημμα κατά πόσο το σύστημα θα έπρεπε να είναι βασισμένο σε μεταγλωττιστή (compiler-based) ή σε διερμηνέα (interpreter-based). Το Mothra ανήκει στην δεύτερη κατηγορία και παρόλη την μακρόχρονη προσπάθεια ανάπτυξης του και την σχετικά ικανοποιητική λειτουργία του σύμφωνα με τους κατασκευαστές του δεν φαίνεται να λειτουργεί σε πρακτικά αποδεκτά χρόνους. Από την άλλη ένα σύστημα που βασίζεται σε διερμηνέα παράγει κώδικα (interpreted code) σαφώς χαμηλότερης απόδοσης από ότι αν ήταν μεταγλωττισμένος (compiled code) [VM98]. Το σύστημα PiSCES παράγει ένα ενιαίο εκτελέσιμο με όλες τις μεταλλάξεις για λόγους πιο αξιόπιστης εκτέλεσης. Επίσης το σύστημα που προτάθηκε να χρησιμοποιεί σχήματα (Mutant Schemas Generator) [UOH93] λόγω της φύσης της ίδιας της προσέγγισης αναγκαστικά απαιτεί να είναι βασισμένο σε μεταγλωττιστή. Βέβαια σε αυτό απαιτείται και ένα σύστημα παρακολούθησης που θα θέτει κάθε φορά τις απαραίτητες μεταβλητές για εκτέλεση της αντίστοιχης μετάλλαξης. Αυτό

όμως αναμένεται να έχει σαφώς καλύτερη απόδοση από το Mothra (αφού δεν φαίνεται να έχει υλοποιηθεί τέτοιο σύστημα από τους συγγραφείς της εργασίας).

Από την άλλη υπήρχε η ιδέα να εφαρμοστεί η μέθοδος των μεταλλάξεων στο byte-code της JAVA. Γενικά σε γλώσσες χαμηλότερου επιπέδου έχουν γίνει λίγα πράγματα όσο αφορά την συγκεκριμένη μέθοδο, χωρίς αυτό να σημαίνει ότι οι βασικές αρχές της δεν μπορούν να εφαρμοστούν και σε αυτό το επίπεδο. Όμως δεν ήταν εύκολο να γίνει κάτι τέτοιο παλιότερα λόγω του μεγάλου πλήθους διαφορετικών γλωσσών μηχανής (assembly languages). Σήμερα ίσως να έχει νόημα να εφαρμοστεί κάτι τέτοιο σε byte-code αφού είναι ανεξάρτητο πλατφόρμας [VM98]. Στην παρούσα προσέγγιση αποφεύχθηκε η μετάλλαξη σε αυτό το επίπεδο για διάφορους λόγους. Η μέθοδος φαίνεται να στηρίζεται στην παραδοχή ότι τα λάθη που είναι πιθανό να κάνει ένας προγραμματιστής είναι λίγα. Σε αυτό όμως υπονοείται ότι μπορεί να τα κάνει. Αν λοιπόν πειράζαμε το πρόγραμμα στο επίπεδο που δεν επεμβαίνει συνήθως ο πραγματικός προγραμματιστής αυτό θα έδινε αποτελέσματα που ίσως να μην είχαν τόση αξία σύμφωνα με τις βασικές αρχές της μεθόδου. Συγχρόνως μελετώντας το διαθέσιμο υλικό διαπιστώθηκε ότι η μέθοδος εμφύτευσης σχημάτων μετάλλαξης θα μπορούσε να επιταχύνει την όλη διαδικασία. Αυτό αμέσως καθιστούσε αρκετά δύσκολη ως αδύνατη την υλοποίηση αντίστοιχου συστήματος σε επίπεδο byte-code. Έπειτα η όλη ιδέα ήταν να μπορεί το σύστημα να αυτό-ελέγχει τα αποτελέσματα από την εκτέλεση κάθε μετάλλαξης. Κάτι τέτοιο κατά την γνώμη μας υλοποιείται ευκολότερα στο επίπεδο της γλώσσας που έχει γραφεί το πρόγραμμα και όχι σε πιο χαμηλό από αυτή. Όλα τα παραπάνω θα γίνουν πιο κατανοητά μετά την παρουσίαση της όλης ιδέας.

5.1. Η ιδέα

«Συναρτήσεις που υπολογίζουν την p(in) και m(in) και αν τοπικά ισχύει

$$p(in) \neq m(in),$$

τότε σκοτώνουν την μετάλλαξη ("killed mutant"). Έπειτα σε εκείνο τον κόμβο δοκιμάζεται κάποια άλλη σχετική μετάλλαξη.»

Αναλυτικότερα σχετικά με τα παραπάνω, έστω το πρόγραμμα p και μια μετάλλαξη αυτού m, τα οποία εκτελούνται με τα ίδια δεδομένα εισόδου in για τα οποία έχουν εξόδους p(in) και m(in) αντίστοιχα. Σύμφωνα με την «αδύναμη» μετάλλαξη αν σε κάποιο κόμβο του προγράμματος το αποτέλεσμα της εκτέλεσης του πρωτότυπου κώδικα είναι διαφορετικό από το αντίστοιχο του μεταλλαγμένου, τότε αυτή η συνθήκη κρίνεται ικανή για να «σκοτώσει» τη μετάλλαξη και τα δεδομένα ικανά για την αποκάλυψη λάθους στον κώδικα. Αν λοιπόν θεωρήσουμε ότι στον κόμβο k εκτελείται ο κώδικας $p_k(in_k)$ και αντίστοιχα $m_k(in_k)$, για την περίπτωση της μεταλλαγμένης έκδοσης, για να θεωρηθεί από την μέθοδο η μετάλλαξη m ως νεκρή ("killed") αρκεί να ισχύει:

$$p_k(in_k) \neq m_k(in_k), \text{ για τον κόμβο k}$$

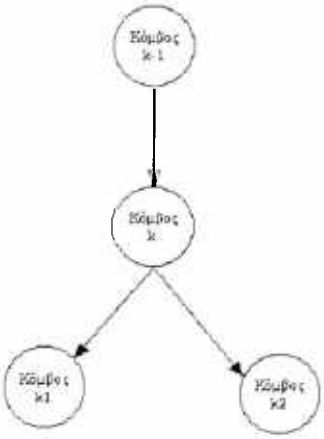
Έστω τώρα ότι στον κόμβο k αντί να εκτελεστεί μόνο ο κώδικας μιας μετάλλαξης έχουμε μεταλλάξει το πρόγραμμα p στο m′ με τέτοιο τρόπο ώστε να εκτελέσει όλες τις N περιπτώσεις των πιθανών μεταλλάξεων εκεί με τα τρέχοντα δεδομένα (που εισήχθησαν στον κόμβο). Κάθε φορά το πρόγραμμα θα χρησιμοποιεί τα ίδια δεδομένα εισόδου στον συγκεκριμένο κόμβο του προγράμματος in_k. Για κάθε τελεστή op_i (operator) της μετάλλαξης θα ελέγχεται πάλι μια συνθήκη ανάλογη της παραπάνω

$$p_k(in_k) \neq m'_k(op_i)(in_k),$$
$$\text{με } op_i \in \{op_1, op_2, op_3, \ldots, op_N\} \text{ στον κόμβο k}$$

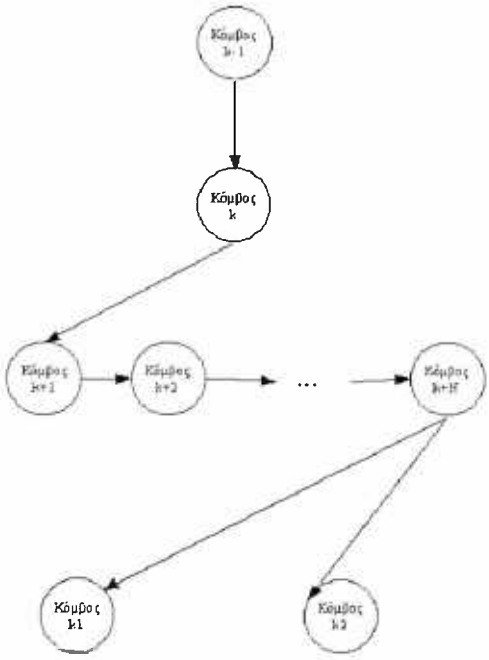
όπου $m'_k(op_i)$ είναι το σημείο στον κόμβο k του «νέου» προγράμματος m′ που εκτελεί τον κώδικα για τον τελεστή op_i.

Προσπαθώντας να απεικονίσουμε την παραπάνω προσέγγιση σε ένα γράφο βλέπουμε την αλλαγή στην δομή της λογικής του προγράμματος στα παρακάτω σχήματα. Η ροή του πρωτότυπου προγράμματος σύμφωνα με την κλασσική εφαρμογή της μεθόδου της μετάλλαξης είναι όπως φαίνεται στο σχήμα αριστερά. Στο σχήμα δεξιά απεικονίζεται ο γράφος του προγράμματος στην περίπτωση που μεταλλαχθεί, ώστε να ελέγχονται όλες οι πιθανές περιπτώσεις για τον ίδιο κόμβο k. Οι κόμβοι k+1 έως k+N είναι αυτοί που προστίθενται στο πρόγραμμα για να ελέγξουν τις μεταλλάξεις του συγκεκριμένου σημείου του κώδικα. Σύμφωνα με

αυτήν την προσέγγιση μετασχηματίζονται συγχρόνως όλοι οι κόμβοι του προγράμματος με τρόπο ανάλογο αυτού για τον κόμβο k.



Σχήμα 10: Μέρος Γράφου Πρωτότυπου προγράμματος



Σχήμα 11: Αντίστοιχο Μέρος Γράφου Μεταλλαγμένης Έκδοσης

5.2. Η meta-mutant συνάρτηση

Έστω ότι έχουμε να μεταλλάξουμε τον παρακάτω κώδικα:

$$X = \alpha + \beta$$

Θα μπορούσαμε να τον αντικαταστήσουμε με μια συνάρτηση που θα παίρνει ως παραμέτρους το αποτέλεσμα της εκτέλεσης του κώδικα του πρωτότυπου προγράμματος σε εκείνο το σημείο, όπως και όλους τους συντελεστές που λαμβάνουν μέρος στη συγκεκριμένη γραμμή κώδικα. Έπειτα θα εκτελεί τον κώδικα για κάθε μετάλλαξη στο εσωτερικό της και ανάλογα κάθε φορά θα σκοτώνει ή όχι την κάθε μία από αυτές, ελέγχοντας το αποτέλεσμα τοπικά με αυτό που θα προέκυπτε αν δεν είχε αλλάξει ο κώδικας. Επίσης για να εκτελεστεί κάθε φορά η τρέχουσα μετάλλαξη θα ελέγχεται η αντίστοιχη παράμετρος από τη διεπαφή του συστήματος ελέγχου,

ώστε να μην εξετάζονται άσκοπα μεταλλάξεις που δεν θα έχει επιλέξει ο χρήστης. Βέβαια αυτό σύμφωνα με την υλοποίηση δεν είναι κάτι που θα διαφοροποιεί ιδιαίτερα το υπολογιστικό κόστος του ελέγχου, γιατί θα αποτρέπει ή όχι την εκτέλεση μιας και μόνο πράξης. Δεν θα σημαίνει πλέον ότι πρέπει να δημιουργηθεί ένα «καινούριο» πρόγραμμα για την συγκεκριμένη μετάλλαξη, το οποίο έπειτα θα πρέπει να μεταγλωττιστεί και να εκτελεστεί, όπως στην κλασσική προσέγγιση της μεθόδου.

Σε εκείνη την προσέγγιση αν δεν κρινόταν αναγκαίο από τον ελεγκτή προσπαθούσε να αποφύγει όσο το δυνατό περισσότερες μεταλλάξεις, ώστε να γλιτώσει κάποιο χρόνο δημιουργώντας μικρότερο πλήθος «λανθασμένων» εκδόσεων του υπό ελέγχου προγράμματος. Η συγκεκριμένη βέβαια προσέγγιση προτρέπει τον έλεγχο όλων των δυνατών μεταλλάξεων, χάρη στο χαμηλό επιπρόσθετο κόστος για κάθε μια περίπτωση. Δεν χρειάζεται να αποφευχθούν κάποιες κατηγορίες μεταλλάξεων, ενώ είναι δυνατή και η χρήση των στατιστικών μετρήσεων με βάση την μελέτη των Offutt και Pan [OP97] για την αξιολόγηση της βαρύτητας κάθε τελεστή στην διαδικασία του ελέγχου με την μέθοδο των μεταλλάξεων. Κάτι τέτοιο θα θύμιζε επιλεκτική (selective) μετάλλαξη.

Αλγοριθμικά η παραπάνω προσέγγιση δεν διαφέρει στην ουσία από την προσέγγιση των Untch, Offutt και Harrold την οποία πρότειναν ως εφαρμογή σχημάτων μετάλλαξης (ομάδες – mutant schemata) [UOH93]. Η παραπάνω διαδικασία ελέγχου όλων των πιθανών μεταλλάξεων ενός κόμβου αλγοριθμικά έχει ως παρακάτω:

1. Αναγνώρισε τα τρία συνθετικά της πράξης στον κόμβο k, δηλαδή τους δύο τελεσταίους και τον ένα τελεστή.
2. Εκτέλεσε τον πρωτότυπο κώδικα και φύλαξε προσωρινά το αποτέλεσμα της πράξης ως res.
3. Θέσε το δείκτη I στην πρώτη από τις πιθανές επιλογές μεταλλάξεων για το συγκεκριμένο σημείο του προγράμματος.
4. Αν έχει επιλεγεί η μετάλλαξη I και είναι διάφορη του πρωτότυπου κώδικα πήγαινε στο βήμα 5, αλλιώς στο βήμα 9.
5. Εκτέλεση της μετάλλαξης I και φύλαξη του αποτελέσματος στο res_m.
6. Σύγκριση των res και res_m. Αν είναι ίσα τότε πήγαινε στο βήμα 7, αλλιώς πήγαινε στο βήμα 8.

7. Η μεταλλαγμένη έκδοση είναι πιθανά ισοδύναμη με την πρωτότυπη. Καταχώρησε την ως τέτοια αν δεν έχεις δεδομένα για παραπέρα έλεγχο της ισοδυναμίας.
8. Η μεταλλαγμένη έκδοση στο εξής θεωρείται «σκοτωμένη» (killed). Καταχώρησε την ως τέτοια.
9. Αν υπάρχει επόμενη επιλογή μετάλλαξης θέσε τον δείκτη I σε αυτή και πήγαινε στο βήμα 5.
10. Επέστρεψε στο πρόγραμμα ως αποτέλεσμα του κόμβου k το res για να συνεχιστεί η εκτέλεση με τον επόμενο κόμβο.

Οι πιθανές επιλογές μετάλλαξης που αναφέρονται στο βήμα 3 μπορούν να οριστούν στατικά και αυτό δίνει την δυνατότητα η meta-mutant συνάρτηση να υπάρχει σε μια βιβλιοθήκη ή σε κάποια κλάση από όπου θα καλείται. Για παράδειγμα για τους αριθμητικούς τελεστές μπορεί να οριστεί ένα σχήμα (αντίστοιχο αυτού στο 7.3). Έτσι το σύνολο των πιθανών μεταλλάξεων είναι αυτό που αντικαθιστά τον τελεστή της πρωτότυπης πράξης με ένα κάθε φορά από τους [+, -, *, /, %] αρκεί να είναι διαφορετικός.

Η ψευδο-κώδικας της συνάρτησης που θα ελέγχει όλες τις μεταλλάξεις για ένα συγκεκριμένο σημείο θα μοιάζει με τον παρακάτω:

5.3. Προσέγγιση 1

```
TypeX ArithOp(enum orig_operator_I, TypeX a, TypeX b, …)  {
TypeX X, Xm;
X = ArithOpTbl[orig_operator_I](a,b);
        For ( int I = 0; I < ARITH_OPP_NUM; I++) {
                if (WeakMutationSet(i)) {
                        Xm = ArithOpTbl[I](a, b);
                        If (X == Xm) EquivalentMutant(i);
                        Else KillMutant(i); }
                        } Return X;}
```

Σχήμα 12: Ψευδό-κώδικας εφαρμογής weak mutation και mutant schemata

Σύμβολο	Επεξήγηση – σχόλια
TypeX	Είναι ο **τύπος** των συντελεστών που λαμβάνουν μέρος στην εκτέλεση της γραμμής κώδικα στο σημείο που μεταλλάσσεται.
Orig_operator_I	Είναι ένα αριθμός που φανερώνει ποια πράξη είναι αυτή που υπάρχει στο πρωτότυπο πρόγραμμα μεταξύ των δύο τελεσταίων (a και b) που δίνονται.
ArithOpTbl	Είναι ο πίνακας συναρτήσεων για κάθε τελεστή.
ARITH_OPP_NUM	Το μέγιστο πλήθος τελεστών. (σταθερή τιμή ανά κατηγορία)
WeakMutationSet	Έλεγχος παραμέτρου για το αν πρέπει να εφαρμοστεί η μέθοδος της «αδύναμης μετάλλαξης» (weak mutation) στον συγκεκριμένο τελεστή I. Αυτό γίνεται με βάση τις επιλογές του χρήστη στην αρχή της εκτέλεσης της διαδικασίας (μέσω της διεπαφής χρήσης του συστήματος).
EquivalentMutant	Η συνάρτηση που ελέγχει την ισοδυναμία ή όχι μιας μετάλλαξης με το πρωτότυπο πρόγραμμα. Μπορεί να είναι **απλή ή πιο ευφυής** σύγκριση αποτελέσματος. Παίρνει σαν είσοδο το αποτέλεσμα του πρωτότυπου κώδικα (X), αλλά και αυτό της μετάλλαξης.
KillMutant	**Με κάποιο τρόπο πρέπει να «σκοτωθεί» η συγκεκριμένη μετάλλαξη**. Μια προσέγγιση είναι να δημιουργείται ένα αρχείο (log file), όπου θα καταγράφονται τα αποτελέσματα του ελέγχου. Εκεί μπορεί να σημειώνει το σύστημα σχετικά με την γραμμή του κώδικα του πρωτότυπου προγράμματος και τον τύπο της μετάλλαξης, που «σκότωσε» ο έλεγχος.

Πίνακας 2: Επεξήγηση προσέγγισης 1

Η αναγνώριση των τριών συνθετικών της πράξης θα γίνεται από το σύστημα γραμματικής ανάλυσης του προγράμματος, το οποίο θα περνάει τα συνθετικά αυτά ως παραμέτρους στις αντίστοιχες meta-mutant συναρτήσεις.

Με την παραπάνω προσέγγιση ο έλεγχος των mutants θα γίνεται ψευδό-παράλληλα και θα μπορεί να συνδυάσει ανάλογα με τις προτιμήσεις του χειριστή θα εκτελείται αδύναμη μετάλλαξη (weak mutation testing). Ο ορισμός της μεθόδου των

μεταλλάξεων προτείνει την εμφύτευση **ενός και μόνο** λάθους σε κάθε μονάδα προγράμματος. Η παραπάνω προσέγγιση είναι όμως ένας συνδυασμός «αδύναμης μετάλλαξης» (weak mutation) [How82, WH88] και της χρήσης «σχημάτων» (mutation schemas) [UOH93].

5.4. Παράδειγμα – Επεξήγηση

```
0) if (A < B) then ...      (πρωτότυπο)
==========================
1) if (A <= B) then ...     (μεταλλάξεις)
2) if (A > B) then ...
3) if (A >= B) then ...
4) if (A == B) then ...
5) if (A != B) then ...
```

Σχήμα 13: Πιθανές μεταλλάξεις μιας συγκριτικής έκφρασης

Όταν λοιπόν φτάσει το πρόγραμμα να εκτελέσει το σημείο, όπου υπάρχει ο κώδικας της (0) αρκεί να εκτελέσει και όλες τις δυνατές περιπτώσεις μετάλλαξης (mutants 1-5) (μέσω μιας **meta-mutant** συνάρτησης) μέχρι να σκοτώσει τις μη ισοδύναμες μεταλλάξεις εκείνου του σημείου σύμφωνα με την «αδύναμη» μετάλλαξη. Με αυτό τον τρόπο γλιτώνουμε την εκτέλεση του κώδικα πριν και μετά το σημείο εκείνο. Σχεδόν είναι σα να μην εκτελούμε όλο το πλήθος των δυνατών mutants, αλλά μόνο έναν **meta-mutant** ή **super-mutant** (λίγο πιο αργό). Σε μεγάλο πρόγραμμα όμως θα σωθεί χρόνος από πολλά βήματα παραγωγής του εκτελέσιμου κώδικα (compilation & linking), όπως και από το χρόνο εκτέλεσης και ελέγχου των αποτελεσμάτων.

5.5. Προσέγγιση 2

Η παραπάνω τεχνική θα μπορούσε ίσως με κάποιες τροποποιήσεις να επιτρέψει και strong ή firm mutation testing. Έστω ότι έχουμε το πρόγραμμα του παρακάτω γράφου:

... → X → X + 1 → X + 2 → X + 3 → ...

Έστω τώρα ότι ο mutant βρίσκεται στον κόμβο (X+1) και μέχρι τον (X) δεν είχε παρατηρηθεί κάποια αλλαγή στα δεδομένα εξόδου από την εκτέλεση κάθε κόμβου (αφού δεν έτρεξε κάποια μετάλλαξη). Έστω ότι στον (X+1) υπάρχει διαφορά λόγω μετάλλαξης. Θα μπορούσε τότε ο κόμβος αυτός να περάσει στον επόμενο και τις δύο εξόδους, ώστε να υπολογιστούν εκεί η μεταλλαγμένη και η γνήσια τιμή να συγκριθούν και στον κόμβο (X+2). Αν εκεί ισχύει p(in) == m(mutated_in), τότε δεν θα υπάρχει λόγος να συνεχιστεί η εκτέλεση του προγράμματος στον κόμβο (X+3), γιατί τα συγκεκριμένα δεδομένα προκάλεσαν λάθος τοπικής ή μεγαλύτερης εμβέλειας αλλά όχι καθολικής. Βέβαια σε αντικειμενοστρεφής γλώσσες, όπως η JAVA, θα έπρεπε να ελεγχθεί η κατάσταση του αντικειμένου (τιμών για κάθε πεδίο) και ίσως και κάθε μετέπειτα σημείο όπου γίνεται αναφορά σε πεδία του. Είναι πιθανό πολύ αργότερα στο πρόγραμμα στον κόμβο X+K να κληθεί διεργασία χρησιμοποιώντας τιμές που δημιουργήθηκαν στον κόμβο X αλλά δεν είχαν χρησιμοποιηθεί μέχρι τότε και να προκαλέσει διαφορετικά αποτελέσματα από αυτά του πρωτότυπου προγράμματος. ➔ Αυτό όμως είναι μια περίπτωση που θα μπορούσε να αντιμετωπιστεί εφαρμόζοντας τις αρχές της συμπαγής ή άκαμπτης μετάλλαξης (firm mutation) [WH88]. Θα έπρεπε να θεωρηθεί δηλαδή ότι από την στιγμή που το πρόγραμμα επανέρχεται σε «σωστή» κατάσταση τα δεδομένα δεν είναι ικανά να αποκαλύψουν λάθος είτε η μεταλλαγμένη έκδοση είναι πιθανά ισοδύναμη με την πρωτότυπη. Ακόμα αν μπορούσε το πρόγραμμα να επιστρέψει και στον κόμβο (X+1) με τα σωστά δεδομένα εισόδου θα μειωνόταν σημαντικά ο χρόνος εκτέλεσης του όλου ελέγχου. Κάτι τέτοιο θα μπορούσε να υλοποιηθεί εύκολα από γλώσσες με χρήση label και goto κλήσεων. Δυστυχώς όμως η JAVA δεν υποστηρίζει αυτά τα στοιχεία και δυσκολεύει την υλοποίηση τέτοιων κύκλων.

Η μορφή (ψευδο-κώδικας) της meta-mutant συνάρτησης σε αυτή την περίπτωση θα μοιάζει ως εξής:

```
TypeX ArithOp(TypeX X, TypeX a, TypeX b, ...) {
TypeX Xm; int I = 0;
while( I < ARITH_OPP_NUM) {
        Xm = ArithOpTbl[I](a, b);
        If (CheckEqual(X, Xm)) KeepMutant(i);
        Else if (WeakMutationSet(i)) KillMutant(i);
        Else Return Xm;
        I++;} Return X;}
```

Σχήμα 14: Εφαρμογή weak ή strong mutation και mutant schemata

Σύμβολο	Επεξήγηση – σχόλια
CheckEqual	Η συνάρτηση που ελέγχει την ισοδυναμία ή όχι μιας μετάλλαξης με το πρωτότυπο πρόγραμμα. Μπορεί να είναι **απλή ή πιο ευφυής σύγκριση** αποτελέσματος. Παίρνει σαν είσοδο το αποτέλεσμα του πρωτότυπου κώδικα (X), αλλά και αυτό της μετάλλαξης. Επίσης, πρέπει να ελέγχει αν έχει επιλέξει ο χρήστης του συστήματος την εφαρμογή της συγκεκριμένης μετάλλαξης ή όχι και ανάλογα να υπολογίζει το αποτέλεσμα της.
KeepMutant	Διαδικασία που κρατάει ζωντανή την μετάλλαξη σε περίπτωση που βρέθηκε ίδιο αποτέλεσμα. Πιθανή περίπτωση ισοδυναμίας που δεν ήταν δυνατό να ελεγχθεί με κάποιο αυτοματοποιημένο τρόπο!
WeakMutation Set	Έλεγχος παραμέτρου για το αν πρέπει να εφαρμοστεί (**τοπικά ή καθολικά**) η μέθοδος της «αδύναμης μετάλλαξης» (weak mutation). Σε αυτό τον έλεγχο το σύστημα θα φτάσει μόνο στην περίπτωση, που το αποτέλεσμα της μετάλλαξης είναι διαφορετικό από ότι το αντίστοιχο του πρωτότυπου προγράμματος και πρέπει να «σκοτωθεί». Αν δεν πρέπει να εφαρμοστεί η αδύναμη μετάλλαξη τότε το πρόγραμμα επιστρέφει την μεταλλαγμένη τιμή. Ένα τέτοιο σύστημα έχει αυξημένη πολυπλοκότητα και είναι αρκετά δύσκολο να υλοποιηθεί. Από την άλλη η χειρότερη περίπτωση είναι να έχουμε ένα μεταλλαγμένο πρόγραμμα με αρκετούς επιπλέον ελέγχους που τελικά κάνει ένα ακόμα πιο δαπανηρό υπολογιστικά έλεγχο σύμφωνα με την «ισχυρή μετάλλαξη» (strong mutation).

Πίνακας 3: Επεξήγηση προσέγγισης 2

Από τα παραπάνω είναι προφανής η αυξημένη πολυπλοκότητα συστήματος που θα εφαρμόζει ταυτόχρονα και δυναμικά «αδύναμη» και «ισχυρή» μετάλλαξη, αλλά και «σχήματα» μεταλλάξεων. Για αυτό λοιπόν στην πρώτη υλοποίηση της προσέγγισης θα αγνοήσουμε την «ισχυρή» μετάλλαξη, εφαρμόζοντας ομαδοποιημένο έλεγχο όλων των επιλεγμένων μεταλλάξεων για κάθε γραμμή κώδικα του πρωτότυπου προγράμματος.

5.6. Συνολική αλγοριθμική περιγραφή

Ο έλεγχος όλου του προγράμματος ακολουθεί τα παρακάτω βήματα.

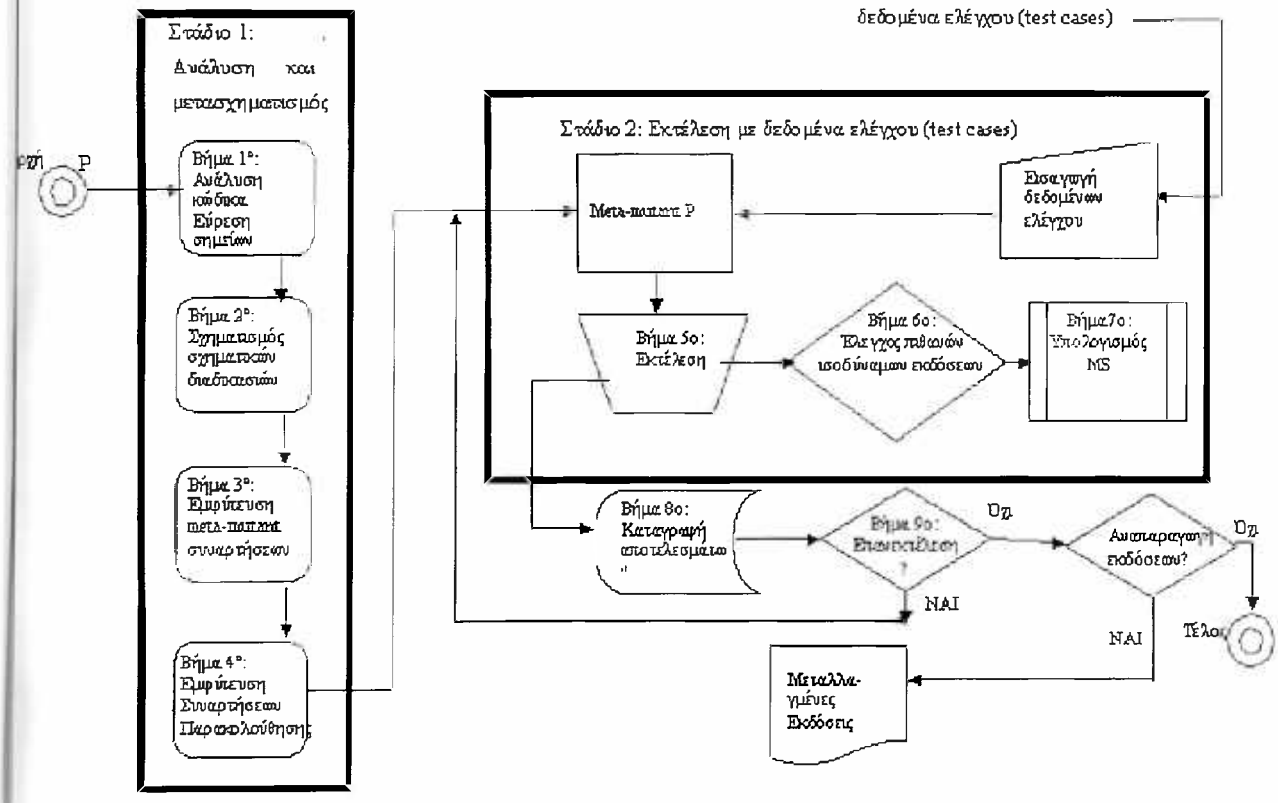
1. Ανέλυσε τον κώδικα και βρες τα σημεία που πρόκειται να μεταλλαχθούν.
2. Σχημάτισε τοπικές συναρτήσεις με τις πράξεις που υπάρχουν εντός των κύκλων (loops) και τις μεταλλάξεις που αναλογούν.
3. Αντικατέστησε της υπόλοιπες πρωτότυπες πράξεις με τις αντίστοιχες «σχηματικές» (schematic) ή «μετά-συναρτήσεις» (meta-mutant procedures). Είναι οι συναρτήσεις που υλοποιούν τον έλεγχο όλων των πιθανών μεταλλάξεων για την συγκεκριμένη πράξη και λαμβάνουν ως παραμέτρους στοιχεία του πρωτότυπου κώδικα, όπως τελεστές (+, -, *, κλπ.) και τελεσταίους (α, 10, 2.6, true, κλπ.) και άλλες χρήσιμες πληροφορίες για το σύστημα ελέγχου (αριθμό γραμμής ή λεξικογραφικά μέρη του κώδικα).
4. Πρόσθεσε κώδικα από τις βοηθητικές κλάσεις, για την διατήρηση πληροφοριών σχετικά με την κατάσταση των μεταλλάξεων κατά την εκτέλεση τους.
5. Εκτέλεσε το μετά- μεταλλαγμένο πρόγραμμα με τα δεδομένα εισόδου που δίνει ο χρήστης. Ταυτόχρονα να γίνεται αυτόματος έλεγχος των πιθανά ισοδύναμων μεταλλαγμένων εκδόσεων κατά την διάρκεια της εκτέλεσης.
6. Αναπαρήγαγε αν ζητηθεί τις υπόλοιπες πιθανές ισοδύναμες εκδόσεις του πρωτότυπου προγράμματος, για να τις αξιολογήσει ο χρήστης.
7. Αφού καθοριστούν οι νεκρές και οι ισοδύναμες μεταλλαγμένες εκδόσεις υπολόγισε το βαθμό αξιοπιστίας αδύναμης μετάλλαξης (Mutation Score – MS).
8. Κατέγραψε τα αποτελέσματα του ελέγχου.
9. Όσο υπάρχουν δεδομένα ελέγχου πήγαινε στο βήμα 5.
10. Τύπωσε τα αποτελέσματα από τους ελέγχους αν ζητηθεί.

Σχήμα 15: Αλγοριθμική περιγραφή του προτεινόμενου συστήματος

Ο παραπάνω αλγόριθμος θα μπορούσε εύκολα να αυτοματοποιηθεί αν ο ελεγκτής δημιουργούσε ένα κύκλο ελέγχων, ένα σύντομο δηλαδή πρόγραμμα, που θα διαβάζει από ένα αρχείο τα δεδομένα ελέγχου και θα φυλάσσει σε κάποιο άλλο τα αποτελέσματα της μετάλλαξης και το βαθμό αξιοπιστίας κάθε συνόλου τέτοιων δεδομένων. Μάλιστα αυτό μπορεί να γίνει με χρήση κάποιου wizard οπότε τον έλεγχο να μπορεί να τον κάνει ο καθένας, χωρίς ιδιαίτερες γνώσεις από την γλώσσα. Αυτό είναι κάτι το οποίο πουθενά δεν φαίνεται να γίνεται αυτόματα και αν όντως γίνεται είναι ίσως αρκετά δύσκολο να υλοποιηθεί, αφού οι μεταλλάξεις εφαρμόζονται σε επίπεδο ενδιάμεσου κώδικα και όχι στην γλώσσα που είναι γραμμένο το πρωτότυπο πρόγραμμα.

Παράλληλα με το παραπάνω σύστημα λειτουργούν τα απαραίτητα υποσυστήματα για την παρακολούθηση του κώδικα και την κατάσταση κάθε μετάλλαξης. Αυτό γίνεται για να υπάρχει η δυνατότητα να αναπαραχθεί ο κώδικας της όποιας μεταλλαγμένης έκδοσης αν αυτό ζητηθεί στο τέλος της διαδικασίας ελέγχου.

Σχηματικά τα παραπάνω φαίνονται στο σχήμα που ακολουθεί.



Σχήμα 16: Προτεινόμενη διαδικασία ελέγχου μεταλλάξεων

5.7. Αξιολόγηση – Κριτική

Κάτι τέτοιο υπάρχει σαν αντίστοιχη ιδέα σε κάποια συστήματα που εφαρμόζουν την αδύναμη εκδοχή της μεθόδου των μεταλλάξεων, όπως για παράδειγμα στο LEONARDO. Αυτό όμως το κάνουν σε επίπεδο γλώσσας μηχανής ή ενδιάμεσου κώδικα και με την βοήθεια ενός άλλου συστήματος που έχει αναλάβει την επίβλεψη της εκτέλεσης του πρωτότυπου προγράμματος, αλλά και όλων των μεταλλαγμένων εκδόσεων του. Επίσης σε εκείνες της προσεγγίσεις δεν είναι ξεκάθαρο το αν επαναλαμβάνεται η εκτέλεση του προγράμματος μέχρι την εκτέλεση της κάθε μετάλλαξης ή σε κάποιες περιπτώσεις αν συνεχίζει μέχρι τέλους από εκείνο το σημείο.

Στην παρούσα προσέγγιση αυτό που προτείνεται είναι να εισαχθούν με την βοήθεια των σχημάτων (schemas) όλες οι δυνατές μεταλλάξεις και να μετασχηματιστεί το ίδιο το πρόγραμμα με τρόπο που θα αναγκάζει την εκτέλεση τους σε εκείνο το σημείο. Αυτό είναι αρκετά πιο εύκολο να υλοποιηθεί σε μια υψηλότερου

επιπέδου γλώσσα λόγω καλύτερης αντίληψης από τους προγραμματιστές. Επίσης είναι πιο εύκολη η συντήρηση και η επέκταση αυτών των κλάσεων ή βιβλιοθηκών που χρησιμοποιούνται από το σύστημα. Θα μπορεί κάποιος που προγραμματίζει στην γλώσσα που είναι γραμμένο το πρόγραμμα που ελέγχεται να ανανεώσει τις meta-mutant συναρτήσεις προσθέτοντας, αφαιρώντας ή γενικότερα τροποποιώντας τις περιπτώσεις μεταλλάξεων ή και ακόμα τις συνθήκες με τις οποίες ελέγχονται οι πιθανά ισοδύναμες εκδόσεις.

Θα μπορούσαν να χρησιμοποιηθούν τεχνικές βελτιστοποίησης κώδικα (compilation optimization techniques) στο βαθμό που αυτό είναι δυνατό, πριν εμφυτευτούν οι meta-mutant συναρτήσεις και αλλάξει τελείως ο κώδικας. Ίσως βέβαια αυτό να είναι κάτι όχι ωφέλιμο αν γίνει γιατί δεν θα επιτρέπει την δημιουργία μεταλλάξεων σε σημεία που θα έχουν χαθεί χάρη της βελτιστοποίησης του κώδικα. Για αυτό θα πρέπει να εφαρμόζεται επιλεκτικά βελτιστοποίηση του κώδικα με την έννοια της απλοποίησης του. Αυτό είναι χρήσιμο όταν χρησιμοποιούνται μεταβλητές με σταθερή τιμή, οπότε πρέπει να εφαρμοστούν και μεταλλάξεις ανάθεσης σε αυτές τις πράξεις. Επίσης, χρήσιμη είναι η απλοποίηση των κύκλων (loops), ώστε να μπορεί το σύστημα γραμματικής ανάλυσης και μετασχηματισμού του κώδικα να δημιουργεί τις LoopCall συναρτήσεις και τις αντίστοιχες μεταλλαγμένες εκδόσεις τους.

Με αυτή την προσέγγιση η δημιουργία των μεταλλάξεων ακολουθεί την κανονική ροή του προγράμματος . Έτσι δεν δημιουργούνται μεταλλάξεις από σημεία του κώδικα που τα δεδομένα δεν μπορούν να φτάσουν. Στην ουσία θα φυτεύονται τα σχήματα με όλες τις πιθανές αλλαγές, αλλά δεν θα μετρώνται αν δεν φτάσει εκεί η εκτέλεση οδηγούμενη από τα δεδομένα ελέγχου. Αυτό σίγουρα μειώνει το πλήθος των μεταλλαγμένων εκδόσεων και κατά συνέπεια αυτών που δημιουργούν ίδιο αποτέλεσμα. Κατά συνέπεια λοιπόν μπορεί να δώσει μεγαλύτερο βαθμό αξιοπιστίας δεδομένων ελέγχου (mutation score).

5.8. Απαιτήσεις – Προδιαγραφές συστήματος

Τα παραπάνω απαιτούν ένα σύστημα γραμματικής και συντακτικής ανάλυσης (parser – "pre-compiler"), ώστε να διαβάζει τον υπό εξέταση κώδικα και να αναλύει το πρόγραμμα στα δομικά του στοιχεία (token). Επίσης θα πρέπει να διατηρεί πληροφορίες σχετικά με το πρόγραμμα ,οι οποίες θα φανούν αργότερα χρήσιμες στον μετασχηματισμό του.

Ταυτόχρονα θα τρέχει ένα σύστημα μετασχηματισμού του προγράμματος που θα συνεργάζεται άμεσα με αυτό της γραμματικής και συντακτικής ανάλυσης και λαμβάνοντας πληροφορίες από αυτό θα εμφυτεύει τις meta-mutant συναρτήσεις από τις κλάσεις που θα αναπτυχθούν σε JAVA (έχοντας μια χρησιμότητα αντίστοιχη με μια βιβλιοθήκη σε C). Έτσι θα χρησιμοποιείται η πλούσια κατά την διάρκεια της μεταγλώττισης πληροφορία. Ακόμα θα δημιουργεί δυναμικά συναρτήσεις που θα περιλαμβάνουν όλες τις κλήσεις/ πράξεις που περιέχονται σε ένα κύκλο (loop) όπως και τις μεταλλαγμένες εκδόσεις αυτών. Μάλιστα θα πρέπει να συμπεριλαμβάνεται η τελευταία κλήση πριν το κύκλο όπως και η πρώτη μετά από αυτόν. Αυτό είναι ένα πιθανό λάθος που πρέπει να ελέγχεται σαν μετάλλαξη και ουσιαστικά υπονοήθηκε από τους Offutt και Lee [OL94] με την περιγραφή του όρου BB-WEAK/N, όπου εξετάζεται το αποτέλεσμα ενός κύκλου που περιέχει μια μετάλλαξη και εξετάζει την επιρροή της αφού τελειώσει η εκτέλεση του και προχωρήσει το πρόγραμμα παρακάτω. Βέβαια στην δημιουργία αυτών των συναρτήσεων για αντικατάσταση των κύκλων είναι πιθανό να δημιουργηθούν συντακτικά λάθη. Αυτά θα πρέπει να τα αποφύγει το σύστημα χρησιμοποιώντας την πληροφορία που θα είναι διαθέσιμη από την γραμματική και συντακτική ανάλυση του προγράμματος, λειτουργώντας σαν κανονικός μεταγλωττιστής για εκείνο το μικρό κομμάτι προγράμματος.

Το σύστημα πρέπει να διατηρεί πληροφορίες για το που βρίσκεται κάθε φορά ο μεταλλαγμένος κώδικας και σε ποιο mutant αντιστοιχεί. Σύμφωνα με την κλασσική προσέγγιση τα συστήματα ελέγχου δημιουργούσαν την αντίστοιχη μεταλλαγμένη έκδοση με αντίστροφη μετάφραση (decompilation) από την ενδιάμεση γλώσσα του διερμηνέα σε αυτή που ήταν γραμμένος ο πηγαίος κώδικας. Το σύστημα που εδώ προτείνεται χρησιμοποιεί υποσυστήματα για την διατήρηση πληροφοριών σχετικών με τον κώδικα, όπως ο αριθμός της τρέχουσας γραμμής και ίχνη κώδικα, που θα βοηθήσουν στην ανακατασκευή των εκδόσεων.

Η υλοποίηση του συγκεκριμένου συστήματος πρέπει να είναι τέτοια, ώστε να βοηθάει την αυτοματοποιημένη εισαγωγή δεδομένων ελέγχου σύμφωνα με τις παραμέτρους κάθε προγράμματος. Το ερώτημα είναι πως θα μπει σε κύκλο ο έλεγχος της meta-mutated έκδοσης με διαφορετικά δεδομένα κάθε φορά. Αυτό που είναι εύκολα υλοποιήσιμο και μπορεί να εφαρμοστεί σε όλες τις περιπτώσεις σαν μια ψευδό-δυναμική προσέγγιση είναι να γράφονται μικρά προγράμματα που θα καλούν με κάποιο επαναληπτικό τρόπο την υπερ-μεταλλαγμένη έκδοση περνώντας δεδομένα

που θα διαβάζονται από κάποιο αρχείο ή κάποια βάση. Η λύση αυτή προϋποθέτει την υλοποίηση από τον ελεγκτή συναρτήσεων ανάγνωσης από την καθορισμένη πηγή δεδομένων, ανάλογα με την μορφή που έχουν καταχωρηθεί εκεί και που απαιτούνται από το πρόγραμμα που ελέγχεται. Επίσης, πρέπει να υλοποιηθούν αντίστοιχες συναρτήσεις αποθήκευσης των αποτελεσμάτων σχετικά με τον έλεγχο και το βαθμό μετάλλαξης. Είναι αδύνατο να οριστούν γενικότερης μορφής τέτοιες συναρτήσεις που να καλύπτουν ποικίλες μορφές (format) αποθήκευσης σε όλες τις δυνατές περιπτώσεις. Για αυτό προτείνεται αυτή η ψευδό-δυναμική προσέγγιση. Έτσι η μορφή των αρχείων από όπου θα διαβάζονται τα δεδομένα ελέγχου θα βοηθάει τον χειριστή του συστήματος ελέγχου να ορίζει τις απαραίτητες συναρτήσεις και προγράμματα, ανάλογα τις παραμέτρους που θα δέχεται σε κάθε κλήση η μονάδα που βρίσκεται υπό έλεγχο. Βέβαια υπάρχει και η εκδοχή η τροφοδότηση με δεδομένα ελέγχου να γίνεται μέσω κάποιου agent σε κάθε κύκλο. Αυτό όμως είναι κάτι που δεν αλλάζει την γενικότερη ιδέα αλλά μονάχα την υλοποίηση της, ενώ φαίνεται να αυξάνει και το κόστος πραγματοποίησης της.

Το σύστημα που θα αναπτυχθεί θα πρέπει να ικανοποιεί κανόνες και προϋποθέσεις όπως περιγράφηκαν σε προηγούμενα συστήματα μετάλλαξης. Συγκεκριμένα θα πρέπει να τηρούνται οι κανόνες του Howden αφού το σύστημα θα εφαρμόζει «αδύναμη μετάλλαξη», όπως και όσο το δυνατό περισσότεροι κανόνες που εφαρμόστηκαν στο Mothra αφού είναι το σύστημα με την μεγαλύτερη ιστορία και ανάλυση από όλα μέχρι σήμερα. Επίσης θα πρέπει να ληφθεί υπόψη η μελέτη αξιολόγησης της αδύναμης μετάλλαξης από τους Offutt και Lee [OL94] και κυρίως για τους κύκλους (loops). Έτσι το σύστημα θα την εφαρμόζει όσο το δυνατόν πιο αξιόπιστα και πιο κοντά στην ισχυρή εκδοχή της. Ακόμα θα πρέπει να δοθούν όσο το δυνατό απαντήσεις στα ερωτήματα που τέθηκαν στην δημιουργία του συστήματος μετάλλαξης για προγράμματα σε Ada [VM98].

Συνολικά τα χαρακτηριστικά του συστήματος ελέγχου με την μέθοδο της εμφύτευσης σχημάτων μετάλλαξης και μερικού ανασχηματισμού και τεμαχισμού του κώδικα θα πρέπει να είναι:

- Γραμματική και Συντακτική Ανάλυση
- Μερική βελτιστοποίηση του κώδικα (προαιρετικά)
- Μετασχηματισμός κώδικα
- Εμφύτευση meta-mutant συναρτήσεων

- Αξιοποίηση πληροφορίας διαθέσιμης από την μετάφραση κατά την εκτέλεση
- Μετάφραση κώδικα (Compiler)
- Δημιουργία μικρών προγραμμάτων για αυτοματοποίηση του ελέγχου με τα υπάρχοντα δεδομένα.
- Εκτέλεση του προγράμματος με τα δεδομένα εισόδου
- Σύγκριση αποτελεσμάτων μετά από εκτέλεση κάθε μετάλλαξης
- Διατήρηση πληροφορίας για την κατάσταση κάθε μετάλλαξης
- Υπολογισμός του βαθμού μετάλλαξης (Mutation Score –MS)
- Ανακατασκευή ολόκληρων μεταλλαγμένων εκδόσεων του προγράμματος

Στο παραπάνω σύστημα δεν προτείνεται οι χρήση αναλυτικών μεθόδων, γιατί κρίνεται ότι δεν είναι δυνατή η απόφανση για την συμπεριφορά μιας μετάλλαξης μόνο με την παρατήρηση της σύνταξης της. Αυτό οφείλεται στο ότι μπορεί να συμπεριφερθεί διαφορετικά κάθε φορά ανάλογα με τα δεδομένα που φτάνουν στο σημείο εκείνο για εκτέλεση, κάτι που είναι αδύνατο να προβλεφθεί σε κάποιες περιπτώσεις αναλύοντας συντακτικά τον κώδικα και μόνο. Κάτι τέτοιο για να επιτευχθεί θα απαιτούσε αρκετή ανάλυση του κώδικα [VM98].

Κεφάλαιο 6ο: Υλοποίηση

Το προτεινόμενο σύστημα έχει σαν βασικό του στόχο την μείωση του πλήθους των μεταλλάξεων. Προς αυτή την κατεύθυνση τροποποιήθηκε κάπως η παραδοχή της μεθόδου για την ικανότητα του προγραμματιστή. Θεωρούμε ότι ο έλεγχος με την μέθοδο των μεταλλάξεων αποτελεί σημαντικό μέρος της διαδικασίας παραγωγής ενός συστήματος αυξημένης κρισιμότητας και με υψηλές απαιτήσεις αξιοπιστίας, όπως για παράδειγμα ένα πρόγραμμα ελέγχου πυρηνικού αντιδραστήρα. Σε τέτοια συστήματα κρίνουμε ότι οι προγραμματιστές δεν είναι άπειροι και κατά συνέπεια δεν είναι πιθανό να κάνουν τα ίδια λάθη με αρχάριους. Αυτό θα μας βοηθήσει παρακάτω που θα ορίζουμε την πολιτική εφαρμογής κάποιων τελεστών μετάλλαξης όπως αυτοί σχεδιάστηκαν από προηγούμενα συστήματα που εφαρμόζουν κάποια παραλλαγή της μεθόδου. Η εμπειρία από προηγούμενες προσπάθειες είναι σημαντικός οδηγός για το νέο σύστημα, ώστε να μην παραληφθούν κρίσιμοι παράμετροι, αλλά και να αποφευχθούν όσο δυνατό προβλήματα.

6.1. Τελεστές μετάλλαξης για την JAVA σύμφωνα με τον Howden

Στην προσπάθεια να οριστούν οι τελεστές μετάλλαξης για ένα σύστημα ελέγχου με την μέθοδο της αδύναμης μετάλλαξης ήταν σημαντικό να ικανοποιηθούν σε πρώτο στάδιο οι κανόνες του Howden που όρισε την συγκεκριμένη προσέγγιση [How82]. Έτσι ορίστηκαν και τέθηκαν σχεδιαστικές αρχές αρχικά για τους παρακάτω τελεστές:

1. **Αναφορά μεταβλητής (Variable reference)**

Το σύστημα κατά την διάρκεια που τρέχει η εκτέλεση του προγράμματος θα διατηρεί δυναμικά πληροφορία με το είδος, τις τιμές και το πλήθος των μεταβλητών όπως αυτές ορίζονται και χρησιμοποιούνται (από το πρόγραμμα). Έτσι όταν εκτελείται μια μετάλλαξη κάποιας έκφρασης, έστω αριθμητικής για δύο τελεσταίους και δεν είναι κάποιος από τους δύο σταθερός (constant), τότε εκτελούνται και όλοι οι πιθανοί συνδυασμοί αλλάζοντας κάθε φορά μια από τις δύο μεταβλητές. Η μεταβλητή που θα χρησιμοποιηθεί θα πρέπει να είναι διαφορετική από τους δύο τελεσταίους και ίδιου τύπου ή κάποιου άλλου με τον οποίο είναι δυνατή η αυτόματη σε χρόνο εκτέλεσης μετάπτωση (run-time casting). Μοναδική εξαίρεση είναι η περίπτωση του

πολλαπλασιασμού. Εκεί έχει νόημα και οι δύο τελεσταίοι να είναι ίδιες μεταβλητές, προσπαθώντας να εκφραστεί δύναμη.

Στον κλασσικό ορισμό της μεθόδου των μεταλλάξεων μάλιστα αναφέρεται ότι κάθε μεταβλητή μεταλλάσσεται μια μόνο φορά σε όλο το πρόγραμμα. Αυτό θα πρέπει να ληφθεί υπόψη από το σύστημα μετάλλαξης και να παρέχει την δυνατότητα για μετάλλαξη κάθε μεταβλητής μια ή περισσότερες φορές στο πρόγραμμα. Η λανθασμένη χρήση μιας μεταβλητής μπορεί να συμβεί σε οποιοδήποτε σημείο του προγράμματος. Για να επιτευχθούν λοιπόν αποτελεσματικότερα δεδομένα ελέγχου είναι καλύτερο να μεταλλάσσεται η χρήση μεταβλητών σε κάθε σημείο που αυτές εμφανίζονται και όχι μόνο την πρώτη φορά. Παρόλα αυτά καλό είναι στο σύστημα ελέγχου να υπάρχει παράμετρος που θα επιτρέπει την επιλογή της μιας ή της άλλης προσέγγισης.

Στην περίπτωση αντικατάστασης ονόματος πίνακα με αυτό κάποιου άλλου, θα πρέπει τα στοιχεία του να είναι του ίδιου τύπου δεδομένων και το μέγεθος του να επιτρέπει την χωρίς διακοπή εκτέλεση του κώδικα στο σημείο μετάλλαξης. Δεν θα έπρεπε δηλαδή να αντικατασταθεί ένας πίνακας από άλλο την στιγμή που η αναφορά στοιχείου του πρώτου θα προκαλούσε κλήση στοιχείου εκτός των ορίων του πίνακα και κατά συνέπεια εξαίρεση (ArrayIndexOutOfBoundsException).

Έστω για παράδειγμα η πρόσθεση των ακεραίων int_a, int_b, ενώ μέχρι εκείνο το σημείο εκτέλεσης έχουν αρχικοποιηθεί οι ακέραιες μεταβλητές int_a, int_b, int_c όπως και οι μεταβλητές τύπου float flt_e, flt_g αλλά και οι τύπου boolean bool_1, bool_2. Οι μεταβλητές int_a και int_b μπορούν να αντικατασταθούν από τις int_a, int_b, int_c, flt_e, flt_g αρκεί να μην ξαναδημιουργηθεί το ίδιο ζευγάρι τελεσταίων. Δεν είναι δυνατή η αντικατάσταση από τις μεταβλητές bool_1 ή bool_2, γιατί δεν ορίζεται αυτόματη μετάπτωση από ακέραια σε λογική τιμή. Αν υπήρχε δήλωση κάποιας μεταβλητής int_d χωρίς όμως να έχει αρχικοποιηθεί πριν την κλήση της παραπάνω γραμμής κώδικα τότε δεν θα μπορούσε να χρησιμοποιηθεί ως αντικατάσταση.

Για να γίνουν όλα τα παραπάνω στην διάρκεια της ανάλυσης του κώδικα το σύστημα μετασχηματισμού εκμεταλλευόμενο την υπάρχουσα πληροφορία για τις δομές του προγράμματος θα μπορούσε να εισάγει κλήσεις σε κάποια ειδική κλάση. Μια περίπτωση αυτής (instance), η διεύθυνση και αρχικοποίηση της θα έχουν επίσης εισαχθεί στο πρόγραμμα στην διάρκεια του μετασχηματισμού. Έτσι η πληροφορία

που υπάρχει διαθέσιμη κατά την μεταγλώττιση θα μπορούσε να χρησιμοποιηθεί και στην διάρκεια της εκτέλεσης του προγράμματος.

Το αναμενόμενο μέγιστο πλήθος μεταλλάξεων που οφείλονται σε αυτό τον τελεστή είναι το άθροισμα των γινομένων των αναφορών σε μεταβλητές ίδιου τύπου t (VR_t) επί το πλήθος των υπόλοιπων δηλωμένων μεταβλητών ($V_t - 2$). Πρέπει να τονιστεί ότι το πλήθος αυτό περιλαμβάνει και τις μεταβλητές άλλου τύπου που η χρήση τους προκαλεί δυναμική μετάπτωση (run-time casting). Ο δεύτερος τελεστής δεν είναι (Vt – 1) επειδή μας ενδιαφέρουν μόνο οι μεταβλητές που είναι μέρος πράξεων και εκεί εξαιρούνται και οι δύο που εμφανίζονται στον πρωτότυπο κώδικα. Συνολικά λοιπόν και λαμβάνοντας όλους τους διαφορετικούς τύπους μεταβλητών έχουμε:

$$\sum_{t=1}^{T}(VR_t \cdot (V_t - 2))$$

2. Ανάθεση μεταβλητής (Variable assignment)

Η μετάλλαξη στην ανάθεση σταθερής τιμής σε μεταβλητή στην εφαρμογή της αδύναμης εκδοχής είναι προφανές ότι δεν μπορεί παρά να οδηγεί πάντοτε σε διαφορετικό αποτέλεσμα ελέγχοντας πάντα το σημείο όπου αυτή πραγματοποιείται. Όταν όμως η τιμή που ανατίθεται σε κάποια μεταβλητή είναι αποτέλεσμα κάποιας έκφρασης (αριθμητικής, σχεσιακής ή δυαδικής), τότε έχει νόημα η μετάλλαξη των παραγόντων της. Μεταφέρεται δηλαδή η μετάλλαξη της ανάθεσης τιμής σε μεταβλητή σε κάθε πράξη που την χρησιμοποιεί.

Για παράδειγμα έστω ότι στην γραμμή κ υπάρχει η αρχικοποίηση με σταθερή τιμή μιας μεταβλητής α, ενώ η πρώτη χρήση της γίνεται στην γραμμή κ+Ν και πριν το τέλος του προγράμματος. Δεν έχει νόημα να ελέγξουμε το αποτέλεσμα αρχικοποίησης της μεταβλητής με διαφορετική σταθερά αμέσως μετά την ανάθεση. Η σταθερά που θα είναι διαφορετική, για να έχει νόημα η μετάλλαξη θα έχει σαφώς προκαλέσει διαφορετικό αποτέλεσμα. Αν όμως όλες αυτές οι μεταλλάξεις για την αρχικοποίηση της μεταβλητής γίνονταν όταν θα δοκιμαζόταν η πρώτη μεταλλαγμένη έκφραση που την χρησιμοποιεί, τότε το αποτέλεσμα θα είχε ενδιαφέρον. Σε εκείνο το σημείο διαφορετική τιμή αρχικοποίησης της μεταβλητής θα είχε περισσότερη αξία ως προς την αξιολόγηση των δεδομένων ελέγχου.

Μεταλλάξεις αυτού του τύπου δημιουργούνται μόνο για αρχικοποίηση μεταβλητών μέσα σε επαναληπτικές δηλώσεις, όπου εκεί εξετάζεται η επιρροή που έχει ένα λάθος στο τέλος του κύκλου.

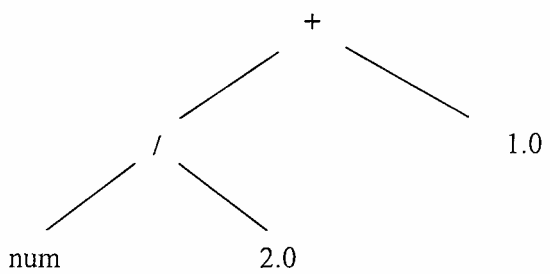
$$\sum_{l=0}^{L}\sum_{t=1}^{T}(IV_t \cdot V_t + 3 \cdot IC_t)_l + 8 \cdot L$$

3. Αριθμητική έκφραση (Arithmetic expression)

Για κάθε αριθμητική έκφραση ορίστηκε μια μέθοδος που εφαρμόζει όλους τους πιθανούς αριθμητικούς τελεστές (+, -, *, /, %) για να δημιουργήσει τις απαραίτητες μεταλλάξεις. Αυτό γίνεται χωρίζοντας σε ζεύγη τους τελεσταίους της κάθε πράξης λαμβάνοντας υπόψη και την προτεραιότητα εφαρμογής κάθε πράξης. Αυτό για να γίνει πρέπει να χρησιμοποιήσει την συντακτική ανάλυση του κώδικα. Έπειτα περνάνε στην συγκεκριμένη μέθοδο μαζί με τον κωδικό της πρωτότυπης μεταξύ τους πράξης, όπως και πληροφορία σχετικά με το αν είναι σταθερές ή μεταβλητές αναφορές. Τα παραπάνω γίνονται από τον μετασχηματιστή που θα συνυπάρχει με το σύστημα συντακτικής και γραμματικής ανάλυσης πριν την μεταγλώττιση του προγράμματος σε byte-code μορφή.

Αναφέρεται παράδειγμα για καλύτερη κατανόηση των παραπάνω. Έστω η πράξη

NewGuess = (num / 2.0) + 1.0;



Σχήμα 17: Συντακτική ανάλυση για NewGuess = (num / 2.0) + 1.0;

Τα ζευγάρια κατά την ανάλυση είναι η μεταβλητή num και η σταθερά 2.0, όπως και το αποτέλεσμα της μεταξύ τους πράξης με την σταθερά 1.0. Έστω ότι προστίθενται δύο φωλιασμένες κλήσεις με σύνταξη κάπως όπως την παρακάτω:

```
NewGuess = ArithWeakOp (
                ArithWeakOp ("num", 2.0, AO_DIV),
                1.0, 2, AO_ADD,);
```

Οι δύο παράμετροι είναι σχετικές με τους τελεσταίους που λαμβάνουν μέρος στην έκφραση. Αν δοθεί το όνομα της μεταβλητής ως αλφαριθμητικό τότε γίνεται και μετάλλαξη αναφοράς μεταβλητής. Στην αντίθετη περίπτωση γίνεται μετάλλαξη της σταθερής τιμής (μείωση ή αύξηση της κατά ένα συντελεστή, περισσότερα αναφέρονται παρακάτω στην παρούσα εργασία), αφού θεωρείται σαν σταθερή τιμή και όχι αναφορά σε κάποια μεταβλητή. Η τρίτη παράμετρος είναι σχετική με την πράξη που πραγματικά εκτελείται στον πρωτότυπο κώδικα. Η μέθοδος ArithWeakOp είναι υπεύθυνη για να εκτελέσει όλες τις πιθανές μεταλλάξεις στο σημείο εκείνο με τις συγκεκριμένες παραμέτρους και να συγκρίνει το αποτέλεσμα τους με αυτό της πρωτότυπης πράξης. Στην διάρκεια της εκτέλεσης της ελέγχεται και η κατάσταση κάθε μετάλλαξης για το αν πρέπει να θεωρηθεί σκοτωμένη ή ισοδύναμη.

Η παραπάνω ανάλυση λαμβάνει ως δεδομένο ότι από την στιγμή που το αποτέλεσμα της μεταλλαγμένης '(num / 2.0)' διαφέρει από της πρωτότυπης θα συνεπάγεται και διαφορετικό αποτέλεσμα στην μετάλλαξη του ίδιου σημείου στην πράξη '(num / 2.0) + 1.0'. Έτσι στην περίπτωση της (num / 2.0) θα ήταν δυνατή και μια διαφορετική ανάλυση της έκφρασης. Κατά την ανάλυση του κώδικα και των πράξεων που υπάρχουν θα μπορούσε να ληφθεί υπόψη και το πρόθεμα της κάθε πράξης (εδώ δεν υπάρχει), όπως και το υπόλοιπο της (εδώ είναι '+ 1.0'). Ο τρόπος αυτός θα είχε ως στόχο τον έλεγχο όλης της πράξης με την μεταλλαγμένη αντίστοιχη της και όχι απλά τον έλεγχο κάποιων μερών της.

Συγκεκριμένα με την πρώτη προσέγγιση αρκεί να είναι διαφορετικά τα (num / 2.0) και (num * 2.0) για παράδειγμα και δεν εξετάζονται πραγματικά τα αποτελέσματα από τις πράξεις '(num / 2.0) + 1.0' και '(num * 2.0) + 1.0', οι οποίες πραγματικά θα επρόκειτο να εκτελεστούν στην συγκεκριμένη μεταλλαγμένη έκδοση. Ας δούμε όμως τις πιθανές περιπτώσεις για το παραπάνω παράδειγμα στον παρακάτω

πίνακα, θεωρώντας τις μεταβλητές A (εδώ 'num / 2.0'), B, M (το αποτέλεσμα της μετάλλαξης του A όταν είναι διαφορετικό) και τον τελεστή op (εδώ '+') μεταξύ τους:

α/α	A	B	op	A op B	M op B	Σχόλια - Εξήγηση
1.1	?	?	+, -	A op B	M op B	Το αποτέλεσμα εδώ είναι πάντα διαφορετικό κατά |M – A|, το οποίο είναι διάφορο του 0, αφού M != A.
1.2	0	!=0	*, /	0	!= 0	Προφανές από βασική αρχή των μαθηματικών.
1.3	?	!=0	%	0	M op B	Εδώ δεν ξέρουμε αν το αποτέλεσμα είναι διαφορετικό. Έστω το παράδειγμα, A = 2, M = 4, B = 2. Τότε (A % B) = (M % B)!
1.4	?	0	*	0	0	Η συνολική πράξη έχει τελικά ίδιο αποτέλεσμα ανεξάρτητα από την σχέση των A,M. Αυτό επειδή το B = 0.
1.5	?	0	/, %	NaN	NaN	Εδώ δεν έχει σημασία η σχέση των A, M γιατί είναι παράνομη η πράξη. Παρόλα αυτά τελικά το αποτέλεσμα είναι το ίδιο συνολικά και θα έπρεπε η δύο εκδόσεις να θεωρηθούν ισοδύναμες αν το B είναι σταθερά ή να δοκιμαστούν με άλλα δεδομένα σε διαφορετική περίπτωση.

Πίνακας 4: Μελέτη περιπτώσεων με φωλιασμένες πράξεις αριστερά

Από τα παραπάνω φαίνεται η ανάγκη σε ορισμένες περιπτώσεις να λαμβάνεται υπόψη και η πράξη στο ακριβώς παραπάνω επίπεδο από την (num / 2.0). Μάλιστα έχει μεγάλη σημασία όταν η πράξη που ακολουθεί είναι *, /, ή % και ειδικά για τις δύο πρώτες στην περίπτωση που ο δεξιός τελεσταίος είναι μηδέν. Βέβαια αν αυτός είναι σταθερά μηδέν, τότε ο μεταγλωττιστής δεν θα το δεχόταν στην περίπτωση των /, %. Πάλι όμως δεν θα μπορούσε να θεωρηθεί ισοδύναμη κάποια μεταλλαγμένη έκδοση αν τα δεδομένα ήταν δυναμικά. Επίσης, δεν είναι σωστό να

μετρώνται «σκοτωμένες» μεταλλάξεις στην περίπτωση που η φωλιασμένη μετάλλαξη δίνει διαφορετικό αποτέλεσμα, ενώ η συνολική όχι (όπως στις περιπτώσεις 1.3,1.4 και 1.5).

Κάτι τέτοιο θα αποδυνάμωνε την μέθοδο και τα αποτελέσματα της. Για αυτό το λόγο τελικά πρέπει να υπάρχουν διαφορετικές κλήσεις ανάλογα με το επίπεδο στο οποίο βρίσκεται η κάθε πράξη. Πρέπει να λαμβάνεται υπόψη η προηγούμενη πράξη μέρος της οποίας είναι αυτή που μεταλλάσσεται, όπως και η θέση του τελεσταίου (αριστερά ή δεξιά από τον τελεστή). Έτσι αν μια οποιαδήποτε πράξη που μεταλλάσσεται είναι αριστερός τελεσταίος μιας πράξης *, /, % και ο δεξιός τελεσταίος είναι σταθερά μηδέν, θα πρέπει να θεωρηθούν οι μεταλλάξεις του χαμηλότερου επιπέδου ισοδύναμες. Αν έτυχε απλά η τιμή να είναι μηδέν στο συγκεκριμένο σενάριο ελέγχου, τότε οι φωλιασμένες μεταλλάξεις δεν θα πρέπει να βαφτίζονται ως ισοδύναμες ή νεκρές, αλλά απλά να μετρώνται. Αν πάλι έχουμε σε πιο πάνω στάδιο από αυτό που ελέγχεται πράξη % με δεξιό τελεσταίο διάφορο του μηδέν θα πρέπει να συγκρίνονται συνολικά οι δύο πράξεις. Στην τελευταία περίπτωση πρέπει να περνάει στο πιο χαμηλό επίπεδο και η τιμή της μεταβλητής που βρίσκεται δεξιά της πράξης %, για να ελέγχεται εκεί το τελικό αποτέλεσμα. Αν μάλιστα υπάρχει μια σειρά από τέτοιες πράξεις καλό θα ήταν να πέρναγε μια λίστα από τους τελεστές.

Αυτό που πρέπει να ελεγχθεί είναι το κατά πόσο είναι κρίσιμη η μελέτη της αντίστοιχης περίπτωσης που η φωλιασμένη πράξη είναι δεξιός τελεστής. Για αυτή την περίπτωση δίνεται ο παρακάτω πίνακας:

α/α	B	A	op	B op A	B op M	Σχόλια - Εξήγηση
2.1	?	?	+, -	B op A	B op M	Το αποτέλεσμα εδώ είναι πάντα διαφορετικό κατά |M – A|, το οποίο είναι διάφορο του 0, αφού M != A.
2.2	!=0	?	*	B op A	B op M	Διαφορετικά αποτελέσματα.
2.3	0	?	*	0	0	Η συνολική πράξη έχει τελικά ίδιο αποτέλεσμα ανεξάρτητα από την σχέση των A,M. Αυτό επειδή το B = 0.
2.4	!=0	!=0	/	B op A	B op M	Διαφορετικά αποτελέσματα.
2.5	0	!=0	/, %	0	0	Πάντα ίδια αποτελέσματα ανεξάρτητα από την μετάλλαξη.
2.6	?	0	/, %	NaN	!NaN	Εδώ το αποτέλεσμα είναι διαφορετικό πάντα, αφού M != 0. Έστω το παράδειγμα, A = 0, M = 4, B = 8. Τότε NaN != (B % M)!
2.7	!=0	!=0	%	B op A	B op M	Εδώ δεν ξέρουμε αν το αποτέλεσμα είναι διαφορετικό. Έστω το παράδειγμα, A = 2, M = 3, B = 18. Τότε (B % A) == (B % M)!

Πίνακας 5: Μελέτη περιπτώσεων με φωλιασμένες πράξεις αριστερά

Τα 2.1 έως και 2.3 ήταν αναμενόμενα λόγω της αντιμεταθετικής ιδιότητας των πράξεων αυτών (+, -, *). Κάθε φορά που η παραπάνω επιπέδου πράξη είναι πολλαπλασιασμός ή διαίρεση (2.3 και 2.5) με σταθερά μηδέν πρέπει να θεωρούνται οι μεταλλάξεις στα πιο χαμηλά επίπεδα ισοδύναμες. Αν δεν είναι σταθερά τότε δεν θα πρέπει να βαφτίζονται ισοδύναμες ή νεκρές, αλλά απλά να μετρώνται μέχρι κάποια δεδομένα να τις σκοτώσουν. Αν πάλι έχουμε σε πιο πάνω στάδιο από αυτό που ελέγχεται πράξη % με αριστερό τελεσταίο διάφορο του μηδέν θα πρέπει να συγκρίνονται συνολικά οι δύο πράξεις. Στην τελευταία περίπτωση (2.7) πρέπει να περνάει στο πιο χαμηλό επίπεδο και η τιμή της μεταβλητής που βρίσκεται αριστερά

της πράξης %, για να ελέγχεται εκεί το τελικό αποτέλεσμα. Αν μάλιστα υπάρχει μια σειρά από τέτοιες πράξεις καλό θα ήταν να πέρναγε μια λίστα από τους τελεσταίους.

Η καλύτερη λύση όμως είναι οι μεταλλάξεις να μετρώνται, όπως και οι καταστάσεις τους να φυλάσσονται κάπου προσωρινά. Έτσι ανεβαίνοντας σε πιο ψηλά επίπεδα της πράξης αν βρεθεί κάποια ειδική περίπτωση να αλλάζουν αναδρομικά οι καταστάσεις όλων των ήδη δημιουργημένων μεταλλάξεων. Για παράδειγμα έστω ότι για μια φωλιασμένη πράξη βρέθηκαν τέσσερις νεκρές μεταλλάξεις. Ανεβαίνοντας όμως στην πιο πάνω πράξη βλέπουμε ότι είναι πολλαπλασιασμός με το μηδέν. Τότε θα πρέπει και οι τέσσερις φωλιασμένες μεταλλάξεις να θεωρηθούν ως «ζωντανές», γιατί τα δεδομένα δεν ήταν ικανά να τις χαρακτηρίσουν ως «νεκρές» ή «ισοδύναμες».

Τέλος από τους αριθμητικούς τελεστές που υπάρχουν στην JAVA κρίθηκε ότι δεν πρέπει να μεταλλάσσονται αυτοί που είναι σύνθετοι και συνήθως χρησιμοποιούνται από έμπειρους προγραμματιστές. Αυτοί οι τελεστές είναι οι {++, --, +=, -=, /=, *=, %=}.

4. Σχεσιακή έκφραση (Relational expression)

Όταν υπάρχει κάποια σχεσιακή έκφραση ο τρόπος εφαρμογής της μετάλλαξης είναι όμοιος με αυτό στην περίπτωση των αριθμητικών εκφράσεων. Πάλι στο πρώτο στάδιο χωρίζεται η πράξη σε ζευγάρια τελεσταίων ανάλογα με την δομή του δένδρου συντακτικής ανάλυσης του κώδικα. Έπειτα καλείται η αντίστοιχη μέθοδος μετάλλαξης στην οποία περνάνε σαν παράμετροι οι δύο τελεσταίοι, ο κωδικός της πρωτότυπης πράξης και πληροφορία σχετικά με το ποια τιμή είναι σταθερή ή αναφορά μεταβλητής. Οι σχεσιακοί τελεστές που εφαρμόζονται είναι (>, >=, <, <=, ==, !=).

Εδώ δεν είναι εύκολο να φανταστεί κάποιος πως μπορεί να είναι μια φωλιασμένη σύγκριση μέσα σε κάποια άλλη. Για αυτό το λόγο η υλοποίηση του σχήματος που θα ελέγχει τις μεταλλάξεις σχεσιακών εκφράσεων είναι πιο εύκολη.

5. Δισήμαντη έκφραση (Boolean expression)

Για τις δυαδικές εκφράσεις οι μεταλλάξεις ελέγχονται με τρόπο ανάλογο των αριθμητικών και των σχεσιακών. Οι τελεστές που ελέγχονται είναι τρεις λογικοί (!, &&, ||) και τρεις δυαδικοί (^, &, |). Σε αυτή την κατηγορία εκφράσεων υπάρχουν κάποιες περιπτώσεις που μπορεί να προκαλέσουν προβλήματα στον χαρακτηρισμό

των μεταλλάξεων με τρόπο αντίστοιχο των αριθμητικών τελεστών *, / και %. Οι πιθανές περιπτώσεις απλά είναι λιγότερες.

Έστω ότι η μητρική λογική πράξη μιας αντίστοιχης φωλιασμένης είναι το λογικό OR με τιμή True ή το λογικό AND με τιμή False σταθερές ή όχι. Θα ήταν ανόητο να γίνει μετάλλαξη στην φωλιασμένη πράξη αν παρουσιάζονταν οι παραπάνω περιπτώσεις με σταθερές τιμές, γιατί τότε κάθε μετάλλαξη θα ήταν ισοδύναμη. Αυτό θα μπορούσε να αποφευχθεί χρησιμοποιώντας την πληροφορία από την γραμματική ανάλυση και αποφεύγοντας την εμφύτευση σχημάτων μετάλλαξης σε τέτοιες περιπτώσεις. Αν πάλι οι πράξεις γίνονται με δυναμικές τιμές τότε θα έπρεπε να προστεθούν και να παραμείνουν ζωντανές μέχρι να βρεθούν καταλληλότερα δεδομένα για να τις σκοτώσουν. Στην τελευταία περίπτωση θα ήταν χρήσιμο αυτό να ήταν γνωστό πριν εκτελεστούν οι φωλιασμένες μεταλλάξεις για οικονομία σε χρόνο. Από την άλλη πάλι το πλήθος των μεταλλάξεων σε αυτή την κατηγορία είναι τόσο μικρό που είναι ανεκτό αυτό το επιπλέον κόστος, πιστεύοντας ότι ο έμπειρος προγραμματιστής δεν χρησιμοποιεί πολλές φωλιασμένες λογικές πράξεις, για χάρη της απλότητας του κώδικα προς καλύτερη κατανόηση του.

Στην περίπτωση των δυαδικών τελεστών αν η μητρική ήταν η πράξη '&' με το μηδέν ή η πράξη ' | ' με το μέγιστο ακέραιο (INT_MAX) και αυτές οι τιμές ήταν σταθερές, τότε οι φωλιασμένες θα έπρεπε να θεωρηθούν ισοδύναμες και να μην δημιουργηθούν καθόλου εξαρχής. Αν οι τιμές αυτές ήταν δυναμικές θα έπρεπε οι φωλιασμένες μεταλλάξεις να ξαναζωντανέψουν μέχρι τα δεδομένα να μην δημιουργούν τέτοιες συνθήκες. Κάτι αντίστοιχο θα έπρεπε να ελεγχθεί και στην περίπτωση που η μητρική πράξη είναι η ' ^ '.

6.2. Τελεστές σύμφωνα με το LEONARDO

Σύμφωνα με την αρθρογραφία το σύστημα LEONARDO [OL94] έχει εφαρμόσει την αδύναμη μετάλλαξη. Για την υλοποίηση του χρησιμοποιήθηκε στην ουσία το σύστημα Mothra, το οποίο σχεδιάστηκε να εφαρμόζει ισχυρή μετάλλαξη [DGK+88, KO91, OC94]. Στο σύστημα που προτείνουμε στην παρούσα εργασία λήφθηκαν υπόψη και αξιολογήθηκαν οι τελεστές μετάλλαξης σύμφωνα με τα LEONARDO και Mothra. Κάποιοι έχουν νόημα ενώ κάποιοι άλλοι κρίθηκε ότι δεν θα βοηθούσαν στην όλη διαδικασία σύμφωνα με την προτεινόμενη σχεδίαση της.

Συγκεκριμένα από την κατηγορία **δηλωτικής ανάλυσης (Statement analysis - sal)** σύμφωνα με τα Mothra και LEONARDO πρέπει να αλλάζει κάθε statement με μια από τις εντολές TRAP, RETURN και CONTINUE, όπως και ο στόχος σε κάθε κλήση GOTO ή DO. Στην JAVA δεν υπάρχει η κλήση GOTO, ενώ η TRAP θα μπορούσε να αντιστοιχηθεί με την catch όπου διαχειρίζεται κάποιο exception (εξαίρεση). Γενικά όμως αυτού του είδους οι τελεστές μετάλλαξης κρίθηκε ότι δεν μπορούν να αποτελούν μέρος του συστήματος που προτείνεται εδώ. Αυτό συμβαίνει γιατί στην εργασία αυτή γίνεται προσπάθεια για περιγραφή ενός συστήματος που θα εφαρμόζει αδύναμη μετάλλαξη. Άρα δεν έχει νόημα να εξεταστούν οι τελεστές αυτοί, αφού είναι σίγουρο ότι θα οδηγούν σε διαφορετικά αποτελέσματα σύμφωνα με τον ορισμό αυτής της εκδοχής της μεθόδου.

Αντιθέτως οι περισσότεροι τελεστές από τις κατηγορίες **αποφαντικής (Predicate analysis - pda)** και **συμπτωματικής ανάλυσης (Coincidental analysis - cca)**, έχουν νόημα για το σύστημα αδύναμης μετάλλαξης με χρήση σχημάτων. Μάλιστα για πολλούς από αυτούς ο τρόπος ελέγχου τους έχει ήδη περιγραφεί παραπάνω σύμφωνα με την κατηγοριοποίηση του Howden. Αναλυτικότερα τώρα:

1. **Εισαγωγή απόλυτης τιμής για κάθε έκφραση.**

Σύμφωνα με το σύστημα Mothra αυτός ο τελεστής μετάλλαξης παίρνει την απόλυτη τιμή και την αρνητική απόλυτη τιμή κάθε έκφρασης. Αυτό για να έχει αξία πρέπει να εφαρμοστεί στο ίδιο σημείο όπου ελέγχεται η ανάθεση μεταβλητής όπως αυτό περιγράφηκε παραπάνω στην προσαρμογή τελεστών μετάλλαξης σύμφωνα με την κατηγοριοποίηση του Howden. Αν για παράδειγμα υπήρχε η μετάλλαξη μιας αριθμητικής έκφρασης σε εκείνο το σημείο μπορεί να δοκιμαστεί το αποτέλεσμα της αν κάθε φορά λαμβανόταν υπόψη η απόλυτη τιμή για κάθε ένα από τους τελεσταίους. Αν έστω είχαμε δηλαδή την

NewGuess = (num / 2.0) + a;

Τότε οι πιθανές μεταλλάξεις σύμφωνα με την εισαγωγή απόλυτης τιμής θα ήταν:

NewGuess = (**abs**(num) / 2.0) + a;

NewGuess = (-**abs**(num) / 2.0) + a;

NewGuess = (num / 2.0) + **abs**(a);

NewGuess = (num / 2.0) + -**abs**(a);

Παρατηρήθηκε πάντως από αρκετές μελέτες ότι δημιουργούνται μεταλλάξεις από αυτό το τελεστή που μπορεί να είναι ισοδύναμες ή να δίνουν ίδια αποτελέσματα ανάλογα τα δεδομένα που εκτελούνται κάθε φορά. Μάλιστα σε κάποια [OP97] προτάθηκε η χρήση περιορισμών, ώστε να μην εφαρμόζεται ο τελεστής απόλυτης τιμής αν είναι θετική η τιμή ή ο αντίστοιχος αρνητικής απόλυτης τιμής αν είναι αρνητική. Εξάλλου η χρήση μιας συνάρτησης δεν μπορεί να είναι πιθανό λάθος ενός προγραμματιστή, ειδικά όταν αυτό επιβάλλει την χρήση βιβλιοθήκης που δεν έχει ήδη συμπεριληφθεί στο πρόγραμμα. Το πιο πιθανό είναι να έχει προσθέσει ή ξεχάσει ένα αρνητικό πρόσημο πριν την συνάρτηση abs. Για αυτό το λόγο αυτό που εφαρμόζεται από το σύστημα που προτείνεται στην παρούσα εργασία, είναι να χρησιμοποιείται η αρνητική τιμή της αναφερόμενης μεταβλητής ή της χρησιμοποιούμενης σταθεράς (αν δεν είναι μηδέν) με κάποιους περιορισμούς:

- Αν πρόκειται να εισαχθεί μετά από αριθμητικό τελεστή πρόσθεσης ή αφαίρεσης δεν χρειάζεται να εφαρμοστεί, γιατί έχει ήδη δοκιμαστεί από την μετάλλαξη του αριθμητικού τελεστή που αποτελεί ισοδύναμη έκδοση. Για παράδειγμα για την πράξη a + b η μετάλλαξη a + (-b) ισοδυναμεί με την a – b.
- Σε οποιαδήποτε άλλη περίπτωση μπορεί να εφαρμοστεί ο παρών τελεστής μετάλλαξης αρκεί να μην δημιουργεί συντακτικό λάθος.

Σημειώνεται ότι δεν υπάρχει λόγος να δημιουργηθεί μετάλλαξη με τον συγκεκριμένο τελεστή για σταθερές, αλλά μόνο για αναφορές σε μεταβλητές και μόνο στην περίπτωση που υπάρχει πράξη μεταξύ δύο τελεσταίων. Δεν έχει δηλαδή αξία να δοκιμαστεί μια μετάλλαξη σε ανάθεση σταθερής ή μεταβλητής τιμής σε μεταβλητή, γιατί είναι σίγουρο ότι η συγκεκριμένη μετάλλαξη θα πρέπει να σκοτωθεί, σύμφωνα πάντα με την αδύναμη προσέγγιση. Για παράδειγμα έστω:

NewGuess = 2.0; Πρωτότυπος κώδικας
NewGuess = -2.0; Διαφορετικό αποτέλεσμα **ΠΑΝΤΑ**

Εξαίρεση αποτελεί η περίπτωση η ανάθεση αυτή να είναι μέσα σε κύκλο, σχετικά με την αντιμετώπιση της οποίας αναφέρθηκαν παραπάνω (βλ. μετάλλαξη ανάθεσης μεταβλητής σύμφωνα με την κατηγοριοποίηση του Howden). Το αναμενόμενο μέγιστο πλήθος μεταλλάξεων που οφείλεται σε αυτό τον τελεστή είναι όσο και το

πλήθος των τελεσταίων που χρησιμοποιούνται σε πράξεις. Το πλήθος αυτό έχει πάνω όριο τον αριθμό αναφορών όλων των μεταβλητών και σταθερών τιμών.

1) Αντικατάσταση σχεσιακών τελεστών

Ο τρόπος ελέγχου μεταλλάξεων αυτού του είδους έχει περιγραφεί παραπάνω στην περιγραφή σχεδιασμού ελέγχου των σχεσιακών εκφράσεων κατά Howden.

2) Αντικατάσταση λογικών τελεστών

Ο τρόπος ελέγχου μεταλλάξεων αυτού του είδους έχει περιγραφεί παραπάνω στην περιγραφή σχεδιασμού ελέγχου των δισήμαντων εκφράσεων κατά Howden.

3) Εισαγωγή μονοσήμαντων (unary) τελεστών

Είναι το μόνο είδος τελεστών μετάλλαξης αυτής της κατηγορίας που δεν εφαρμόζεται από το προτεινόμενο σύστημα.

4) Τροποποίηση ανάθεσης τιμών

Ο τρόπος ελέγχου μεταλλάξεων αυτού του είδους έχει περιγραφεί παραπάνω στην περιγραφή σχεδιασμού ελέγχου ανάθεσης μεταβλητής κατά Howden.

5) Τροποποίηση σταθερών παραμέτρων

Η τροποποίηση σταθερών παραμέτρων εφαρμόζεται με τρόπο αντίστοιχο της τροποποίησης ανάθεσης τιμών. Όταν ελέγχονται οι μεταλλάξεις με αντικατάσταση αριθμητικού τελεστή κάποιας πράξης, σε εκείνο το σχήμα οι τελεσταίοι που είναι σταθερές τιμές θα τροποποιούνται κατά κάποιο σταθερό παράγοντα ανάλογα με το είδος της κάθε σταθεράς. Οι παράγοντες που επιλέχθηκαν σύμφωνα με το σύστημα μετάλλαξης προγραμμάτων σε Ada83 [VM98] είναι ανάλογα με το τύπο τους:

Τύπος	Τροποποίηση
Ακέραιος	+1 και −1.
Float (πραγματικός)	*1.05 και *0.95
Χαρακτήρας	Προηγούμενος και επόμενος.

Στην περίπτωση που υπάρχει μια διαίρεση πιστεύεται ότι ένας προγραμματιστής δεν θα διαιρούσε ποτέ οποιαδήποτε παράσταση με το μηδέν ή το ένα. Έτσι στην περίπτωση των ακεραίων είναι σίγουρο ότι δεν θα δημιουργηθεί ποτέ διαίρεση με το μηδέν, ενώ δεν θα πρέπει να εκτελείται διαίρεση με το ένα (αν αυτή προκύψει λόγω μετάλλαξης). Επίσης στην περίπτωση των πραγματικών αριθμών είναι σίγουρο ότι πολλαπλασιάζοντας με αυτούς τους παράγοντες, δεν θα εμφανιστεί το μηδέν ως διαιρέτης. Για να συμβεί αυτό θα πρέπει το υπό έλεγχο πρόγραμμα να έχει ήδη το μηδέν σταθερά ως διαιρέτη. Αυτό όμως δεν θα το επέτρεπε ο μεταφραστής της γλώσσας.

Το μέγιστο αναμενόμενο πλήθος αυτών των μεταλλάξεων είναι ίσο με το διπλάσιο του αθροίσματος των χρήσεων σταθερών τιμών σε πράξεις του προγράμματος.

6) **Συμπτωματική ανάλυση** (Coincidental analysis - cca) **ή** αντικατάσταση **μεταβλητών**

Στα Mothra και LEONARDO περιγράφεται μετάλλαξη με αντικατάσταση κλιμακούμενων μεταβλητών (scalar variables), αναφορών σε πίνακες και σταθερών τιμών (constants) από άλλες κλιμακούμενες μεταβλητές, αναφορές πινάκων και σταθερές τιμές. Αντικατάσταση ονόματος πίνακα από όνομα κάποιου άλλου. Όλα αυτά στο σύστημα που εδώ προτείνεται αντιστοιχούν στην μετάλλαξη αναφοράς μεταβλητής σύμφωνα με την κατηγοριοποίηση του Howden και ο τρόπος υλοποίησης της περιγράφηκε παραπάνω.

7) **Αντικατάσταση αριθμητικών τελεστών**

Σε αυτή την κατηγορία σύμφωνα με τους δημιουργούς των συστημάτων αυτών ανήκει και η αντικατάσταση αριθμητικών τελεστών. Ο τρόπος ελέγχου μεταλλάξεων αυτού του είδους έχει περιγραφεί παραπάνω στην περιγραφή σχεδιασμού ελέγχου των αριθμητικών εκφράσεων κατά Howden.

8) **Αντικατάσταση σταθερών**

Στο προτεινόμενο σύστημα δεν κρίνεται απαραίτητη η υποστήριξη αντικατάστασης σταθερών (constants) από αναφορές μεταβλητών. Κάτι τέτοιο δεν είναι πιθανό λάθος από ένα έμπειρο προγραμματιστή. Επίσης κάθε μεταβλητή, όπως

περιγράφηκε σε πιο πάνω σημείο της εργασίας, πρέπει να αντικαθίσταται από κάθε μια άλλη ίδιο τύπου ή τύπου με τον οποίο μπορεί σε χρόνο εκτέλεσης να γίνει μετάπτωση (run-time casting).

6.3. Τελεστές σύμφωνα με το σύστημα της NASA για Ada83

Στην προσπάθεια διαμόρφωσης των προδιαγραφών ενός συστήματος μετάλλαξης για προγράμματα JAVA λήφθηκαν υπόψη και οι τελεστές που χρησιμοποιήθηκαν σε αντίστοιχο σύστημα για προγράμματα σε γλώσσα Ada [VM98]. Οι κατηγοριοποίηση των τελεστών στο συγκεκριμένο σύστημα, που αναπτύχθηκε με την χορηγία της NASA και την επίβλεψη του Offutt, χώριζε τους τελεστές μετάλλαξης σε πέντε κλάσεις:

1. **Αντικατάστασης τελεσταίων** (operands)

Οι τελεσταίοι που αντικαθίστανται στην Ada είναι μεταβλητές, σταθερές, αναφορές σε πίνακες, αναφορές σε εγγραφές και δείκτες. Οι τέσσερις πρώτες κατηγορίες έχουν καλυφθεί από την αντικατάσταση μεταβλητής όπως περιγράφηκε πιο πριν σύμφωνα με την κατηγοριοποίηση κατά Howden. Όσο αφορά τους τελεστές μετάλλαξης για δείκτες υπενθυμίζεται ότι δεν υπάρχει η έννοια του δείκτη στην JAVA. Άρα δεν έχει νόημα να υλοποιηθούν.

Στο σύστημα για την Ada υπάρχουν μερικοί ακόμα τελεστές μετάλλαξης. Υπάρχει τελεστής εξαφάνισης της αρχικοποίησης μιας μεταβλητής, το οποίο αν εφαρμοζόταν στη JAVA δεν θα ήταν αποδεκτό από το μεταγλωττιστή της. Επίσης για την Ada υπάρχουν δύο τελεστές αντικατάστασης ονομάτων εγγραφών (record) ή πεδίων τους όταν το πεδίο που καλείται υπάρχει και στις δύο ή όταν η αναφορά στο νέο πεδίο που εισάγεται υπάρχει στην δομή της εγγραφής με κατάλληλο τύπο αντίστοιχα. Αν και αυτοί οι τελευταίοι τελεστές μετάλλαξης ίσως να μπορούσαν να προσαρμοστούν και στην JAVA αντιστοιχίζοντας τις εγγραφές με κλάσεις, η προσέγγιση όμως που ακολουθήθηκε δεν επέτρεψε τέτοιου είδους μετάλλαξη.

Χρήσιμες φάνηκαν μερικές οδηγίες που υπάρχουν για το σύστημα σε Ada83 σχετικά με την μετάλλαξη τελεσταίων και θα είχαν νόημα σε ένα σύστημα αδύναμης μετάλλαξης. Είναι χρήσιμη η μετάλλαξη:

- αρχικοποιήσεων και

- αναφορών σε απαριθμημένους τύπους (αν και αυτού του είδους οι τύποι δεν υποστηρίζονται από την JAVA),
- των αντικειμένων με τύπο CONSTANT - αντίστοιχα στην JAVA ΤΥΠΟΥ final - (στο σύστημα που εμείς προτείνουμε αντικείμενα αυτού του τύπου θεωρούνται σταθερές τιμές και τροποποιούνται την στιγμή της χρήσης τους και όχι στον ορισμό τους),
- παράμετροι κύκλων (loop)
- εξωτερικών μεταβλητών

Από την άλλη δεν πρέπει να μεταλλάσσονται:

- τύποι δεδομένων
- δηλώσεις μεταβλητών (declarations)
- οι σταθερές στις δηλώσεις case (αντίστοιχες switch στην JAVA)
- οι παράμετροι σε κλήσεις for, γιατί θεωρούνται δηλώσεις (declarations)

2. **Τελεστές μετάλλαξης δηλώσεων**.

Για την Ada83 υποστηρίζονται 13 τελεστές μετάλλαξης δηλώσεων. Κάποιοι από αυτούς δεν μπορούν να εφαρμοστούν στο μοντέλο της αδύναμης μετάλλαξης, όπως κλήσεις GOTO, EXIT (δεν υπάρχουν στην JAVA), RETURN και RAISE (κλήση throw στην JAVA). Για να βρεθεί η διαφορά μιας τέτοιας μετάλλαξης θα έπρεπε να ελεγχθεί το συνολικό αποτέλεσμα μετά την ολική εκτέλεση του προγράμματος. Χρήσιμες μεταλλάξεις που μπορούν να εφαρμοστούν στο σύστημα που εμείς αναπτύσσουμε είναι:

- **Αντικατάσταση του while από do-while και αντίστροφα**.
- Μετάλλαξη στο **πλήθος εκτέλεσης κύκλου** (loop). (Αυτό περιγράφεται αναλυτικότερα παρακάτω.)
- **Μετάλλαξη στις επιλογές ενός switch-case**. Κρατώντας σταθερή την έκφραση που ελέγχεται, αλλάζουμε κάθε φορά μια από τις δυνατές περιπτώσεις. (Περισσότερες λεπτομέρειες υπάρχουν σε παρακάτω σημείο του ίδιου κεφαλαίου.)

3. **Τελεστές μετάλλαξης εκφράσεων**

Οι περισσότεροι από τους 14 τελεστές της συγκεκριμένης κατηγορίας έχουν ήδη περιγραφεί με βάση την κατηγοριοποίηση του Howden ή στη σύγκριση με το

σύστημα LEONARDO. Δεν υπάρχει έτσι λόγος να αναλυθούν παραπάνω περιπτώσεις μετάλλαξης απόλυτης τιμής, αριθμητικών/ λογικών/ σχεσιακών και unary εκφράσεων. Το «στριφογύρισμα» πεδίου (domain twiddle), όπως αναφέρεται από το σύστημα για την Ada83, έχει περιγραφεί πιο πριν ως μετάλλαξη των σταθερών τιμών.

Από τους 14 τελεστές όσοι ελέγχουν την δημιουργία εξαιρέσεων δεν έχουν λόγο ύπαρξης σε ένα σύστημα αδύναμης μετάλλαξης. Έτσι δεν υπάρχουν στο σύστημα που προτείνεται για την JAVA τελεστές που προκαλούν exception για δεδομένα ίσα με το μηδέν ή σε περιπτώσεις υπερχείλισης (overflow) ή «υποχείλισης» (underflow). Επίσης κρίθηκε ότι δεν μπορεί να είναι αρκετά πιθανό από ένα προγραμματιστή ένα λάθος κατά την κλήση υποπρογράμματος και για αυτό το λόγο δεν προτείνονται αντίστοιχοι τελεστές μετάλλαξης, παρόλο που υπάρχουν στο σύστημα για Ada83. Παρόλα αυτά, περιγράφεται παρακάτω ο τρόπος που αυτό θα μπορούσε να υλοποιηθεί στο σύστημα που προτείνεται από την παρούσα εργασία.

4. **Τελεστές μετάλλαξης κάλυψης**

Είναι 4 τελεστές που αναγκάζουν την κάλυψη τουλάχιστον μια φορά κάθε δήλωσης και κάθε έκφρασης στο πρόγραμμα. Κάτι τέτοιο είναι πέρα από τα όρια της μεθόδου μεταλλάξεων για αξιολόγηση δεδομένων ελέγχου, όπως αυτή ορίζεται στην πλειοψηφία της αρθρογραφίας. Έτσι το σύστημα για την JAVA δεν σχεδιάστηκε για έλεγχο κάλυψης και κατά συνέπεια δεν λήφθηκαν υπόψη οι τελεστές αυτής της κατηγορίας.

5. **Ανάθεσης εργασιών προγράμματος (tasking)**

Είναι 3 ειδικοί τελεστές μετάλλαξης που εξαρτώνται από τα χαρακτηριστικά της γλώσσας. Αφού λοιπόν δεν υπάρχουν αντίστοιχα χαρακτηριστικά στην JAVA δεν λήφθηκαν υπόψη στο προτεινόμενο σύστημα για την JAVA.

6.4. Προδιαγραφές σύμφωνα με το Mothra

Παρακάτω παρουσιάζονται οι κανόνες που πρέπει να τηρούνται, σύμφωνα με τους King και Offutt [KO91] δημιουργούς του Mothra, από κάθε σύστημα ελέγχου προγραμμάτων με την μέθοδο των μεταλλάξεων.

1. Εύκολα μεταλλάξιμος ενδιάμεσος (intermediate) κώδικας.

2. Το σύστημα ελέγχου πρέπει να παρακολουθεί από κοντά την συμπεριφορά εισόδου/ εξόδου του υπό έλεγχο προγράμματος.
3. Πρέπει να είναι δυνατή η προς τα πίσω μετάφραση του ενδιάμεσου κώδικα προς το πρωτότυπο πρόγραμμα αλλά και την μεταλλαγμένη έκδοση αυτού.
4. Το πρόγραμμα που ελέγχεται δεν πρέπει να προκαλεί αποτυχία συστήματος (system failure)
5. Επειδή ο διερμηνέας πρέπει να εκτελεί μεγάλο πλήθος προγραμμάτων θα πρέπει να βελτιστοποιεί την ταχύτητα του όποτε δυνατόν.
6. Ο διερμηνέας θα πρέπει να χειρίζεται ένα σεβαστό υποσύνολο της γλώσσας όπου είναι γραμμένα τα υπό εξέταση προγράμματα.

Από τα παραπάνω σημειώνουμε ότι στην προσέγγιση που προτείνεται από την παρούσα εργασία δεν υπάρχει «ενδιάμεσος» κώδικας με την έννοια που εμφανίζεται στο σύστημα Mothra. Παρόλα αυτά θα μπορούσαμε να θεωρήσουμε ως «ενδιάμεσο» τον κώδικα που υπάρχει στην μια υπερ-μεταλλαγμένη έκδοση του προγράμματος που ελέγχεται. Ο **κανόνας 3** λοιπόν ικανοποιείται από το σύστημα μας. Ο **κανόνας 5** που αναφέρεται στην βέλτιστη ταχύτητα εκτέλεσης των μεταλλαγμένων προγραμμάτων, ικανοποιείται γιατί το υπερ-μεταλλαγμένο πρόγραμμα που τρέχουμε έχει βελτιστοποιηθεί από τον μεταγλωττιστή της γλώσσας κατά την διαδικασία παραγωγής του byte-code. Όσο για τον **κανόνα 6** έχοντας ένα μεταγλωττιστή της γλώσσας είναι σίγουρο ότι χειριζόμαστε ένα σεβαστό υποσύνολο της, αλλιώς δεν θα μπορούσε να παράγει εκτελέσιμο κώδικα. Τώρα όσο αφορά τον **κανόνα 1** το κατά πόσο είναι εύκολη ή δύσκολη η μετάλλαξη ενός προγράμματος δεν αποτελεί μέτρο στη δική μας προσέγγιση. Αυτό γιατί στην ουσία ένα τέτοιο σύστημα σαν αυτό που προτείνεται από την παρούσα εργασία, αφού δημιουργηθεί θα λειτουργεί με τρόπο διαφανή για τον ελεγκτή και κατά συνέπεια με τρόπο εύκολο.

Οι δυσκολίες που εμφανίζονται στην δική μας προσέγγιση έχουν να κάνουν με τους κανόνες 2 και 4. Αρχικά ο **κανόνας 2** είναι κάτι που δεν έχει πλήρη αντιστοιχία ανάμεσα στο Mothra και το σύστημα που εδώ προτείνεται. Αυτό γιατί στην ουσία όλος ο έλεγχος γίνεται από το ίδιο το πρόγραμμα που ελέγχεται. Σε επίπεδο υλοποίησης ίσως θα ήταν αρκετό να υπάρχει μια κλάση (JMSTrace) που θα παρακολουθεί τα δεδομένα και θα καταγράφει εισόδους και εξόδους του προγράμματος. Υπενθυμίζεται δε ότι το Mothra χρησιμοποιεί «δυνατή» μετάλλαξη

για έλεγχο των προγραμμάτων, ενώ το σύστημα που προτείνεται από την παρούσα εργασία εφαρμόζει κυρίως «αδύναμη».

Όσο για τον **κανόνα 4** αυτό που φαίνεται σαν πιθανότερο ενδεχόμενο είναι να εμφανιστούν προβλήματα από ατελείωτους κύκλους εκτέλεσης σε κάποια μεταλλαγμένη έκδοση του πρωτότυπου κώδικα. Κάτι τέτοιο θα πρέπει να ελεγχθεί στο βαθμό που αυτό γίνεται στο επίπεδο της δημιουργίας των MutatedLoopCalls από τον μεταγλωττιστή, αν χρησιμοποιούνται για παράδειγμα μεταβλητές στην συνθήκη του κύκλου που δεν έχουν αρχικοποιηθεί ή αν δεν παρατηρείται αλλαγή της επιστρεφόμενης τιμής αυτής λόγω σταθερότητας των τιμών που λαμβάνουν μέρος σε αυτή. Επίσης, αυτό που μπορεί να γίνει είναι κατά την εκτέλεση του πρωτότυπου κύκλου να φυλάσσεται το πλήθος των επαναλήψεων με σκοπό να χρησιμοποιηθεί με κάποια ανοχή στις μεταλλαγμένες εκδόσεις του. Όταν οι επαναλήψεις μιας μετάλλαξης αυξάνονται κατά πολύ πέρα από αυτό τον αριθμό τότε θα πρέπει να σταματάει η εκτέλεση της. Σε αυτή την περίπτωση η μετάλλαξη θα πρέπει να θεωρηθεί είτε «σκοτωμένη» είτε σαν ατέρμονη και να αναφερθεί ανάλογα (να βγαίνει κάποιο σχετικό μήνυμα ή να φυλάσσεται για να κριθεί από τον ελεγκτή). Άλλη προσέγγιση σε αυτό το πρόβλημα είναι να περνά στην «κυκλική» συνάρτηση μια λίστα με τα στιγμιότυπα σε κάθε κύκλο της επιστρεφόμενης (υπολογιζόμενης) τιμής. Έτσι αν σε κάποιο κύκλο βρεθεί ότι είναι το προσωρινό αποτέλεσμα ίδιο με το αντίστοιχο στιγμιότυπο τότε μπορεί να σταματήσει ο κύκλος και να θεωρηθεί πιθανή ισοδύναμη μετάλλαξη η συγκεκριμένη έκδοση του προγράμματος.

6.5. Μετάλλαξη κλήσεων σε μέθοδο

Η μετάλλαξη αυτού του τύπου διακρίνεται σε δύο υποπεριπτώσεις. Η πρώτη είναι να αλλάξει η μέθοδος με κάποια άλλη από την ίδια κλάση, ενώ η άλλη πιθανότητα είναι να κληθεί μέθοδος με το ίδιο όνομα και χαρακτηριστικά που ανήκει όμως σε άλλη κλάση. Αυτό θα είχε νόημα αν η καλούμενη μέθοδος επέστρεφε κάποια τιμή. Σε αυτή την περίπτωση αυτό που θα μπορούσε να γίνει είναι στο επίπεδο της μεταγλώττισης και μετάλλαξης του όλου προγράμματος να φυτευτούν σχήματα, όπου θα καλούνται όλες οι δυνατές παραλλαγές και μέσα στα οποία θα ελέγχονται τα αποτελέσματα τους. Οι επιλογές θα καθορίζονται από τις παραμέτρους του καλούμενου υποπρογράμματος και την επιστρεφόμενη τιμή. Θα πρέπει να ταιριάζουν οι παράμετροι στο πλήθος τους και στην σειρά που οι τύποι τους

εναλλάσσονται. Επίσης θα πρέπει η επιστρεφόμενη τιμή να είναι του ίδιου τύπου ή κάποιου ο οποίος μπορεί να μεταπέσει δυναμικά σε χρόνο εκτέλεσης σε κάποιον επιτρεπτό σύμφωνα με την έκφραση που μεταλλάσσεται στο σημείο εκείνο. Μάλιστα θα μπορούσε να επιλέξει ο μετασχηματιστής για το αν θα ψάξει τις αναγκαίες επιλογές από τις κλάσεις που χρησιμοποιούνται ήδη στο πρόγραμμα ή από όλες τις παρεχόμενες βιβλιοθήκες.

Μια δεύτερη προσέγγιση είναι να ελέγχονται οι καταστάσεις των κλάσεων, μετά από την εκτέλεση της κάθε μεθόδου, ώστε να φανερωθεί διαφορά ακόμα και στην περίπτωση που τα επιστρεφόμενα δεδομένα είναι ίδια ή ακόμα και όταν δεν υπάρχουν (συναρτήσεις που δεν επιστρέφουν τιμή). Αυτή η διαδικασία είναι σχετικά περίπλοκη καθότι θα πρέπει να ελέγχεται κάθε πεδίο από την κλάση της οποίας μέλος είναι η καλούμενη μέθοδος (υποπρόγραμμα). Αυτό που θα μπορούσε να γίνει είναι να φυλάσσεται ένα αντίγραφο της μνήμης ή έστω του instance της κλάσης για να αποτελεί κριτήριο στον μετέπειτα έλεγχο. Βέβαια δεν θα είχε νόημα αυτός ο έλεγχος σε περίπτωση που η καλούμενη μέθοδος ανήκε σε διαφορετική κλάση από αυτή που ανήκε η μεταλλαγμένη κλήση.

6.6. Μετάλλαξη κλήσεων πεδίου

Για αυτή την κατηγορία μεταλλάξεων ισχύει κάτι αντίστοιχο με τις μεταλλάξεις των κλήσεων μεθόδων. Χωρίζεται σε δύο υποκατηγορίες. Η πρώτη είναι να αλλάξει το πεδίο που αναφέρεται με άλλο ίδιου τύπου από την ίδια κλάση. Η άλλη πιθανότητα είναι να κληθεί πεδίο με το ίδιο όνομα και χαρακτηριστικά που ανήκει όμως σε άλλη κλάση. Σε όποια από τις δύο περιπτώσεις αυτές οι κλήσεις θα πρέπει να γίνουν από τον μετασχηματιστή στην διάρκεια της μεταγλώττισης, όπου θα είναι γνωστές όλες οι άλλες κλάσεις, όπως και τα πεδία με τους τύπους τους. Εδώ ο μετασχηματιστής θα πρέπει να ψάξει τις αναγκαίες επιλογές από τις κλάσεις που χρησιμοποιούνται ήδη στο πρόγραμμα και όχι από όλες τις δυνατές βιβλιοθήκες. Το τελευταίο δεν έχει νόημα γιατί δεν θα υπάρχουν δεδομένα στην μνήμη από instance κάποιας εξωτερικής κλάσης. Ο τρόπος υλοποίησης των αναγκαίων ελέγχων για αυτή την κατηγορία μετάλλαξης θα γίνεται μέσα από ένα δυναμικά δημιουργούμενο σχήμα.

6.7. Μετάλλαξη των κύκλων (Loop)

Αρχικά υπήρξε μια ιδέα κάθε κύκλος (loop) να μετατρέπεται σε ένα κοινό μεταλλαγμένο (mutated). Αυτό ίσως να βοηθούσε σε καλύτερη διαχείριση του ελέγχου του θεωρώντας τις αντίστοιχες μορφές για κάθε περίπτωση επαναληπτικών κλήσεων (iterative calls ή loop). Στην πρώτη να δημιουργείται μια προ-μεταλλαγμένη έκδοση όπως φαίνεται στον παρακάτω πίνακα σύμφωνα με την γραμματική σύνταξη κάποιων γνωστών loop:

Πρωτότυπο	Προ-μεταλλαγμένο
Do {	Statement*
Statement*	While (Condition+) {
} While (Condition+)	Statement*
	}
While (Condition+) {	While (Condition+) {
Statement*	Statement*
}	}
for(statement1*;Condition*; statement2*)	statement1*
{	While (Condition+) {
Statement3*	Statement3*
}	Statement2*
	}

Πίνακας 6: Ενοποιημένος μετασχηματισμός επαναληπτικών δηλώσεων

Άρα το πρόβλημα ανάγεται στην μετάλλαξη της κλήσης του while loop με την γενική μορφή:

```
While (Condition+) {
        Statement*
}
```

Αυτό όμως δεν θα επέτρεπε την αντικατάσταση του while από do-while και αντίστροφα κάτι το οποίο αποτελεί τελεστή μετάλλαξης σε άλλα συστήματα. Ακόμα και η κλήση for πρέπει να διατηρηθεί, ώστε να μεταλλαχθούν οι παράμετροι ορισμού της.

Έτσι σε πρώτη φάση δημιουργίας της προ-μετάλλαξης θα μεταφέρονται σε συνάρτηση όλες οι κλήσεις Statement* που υπάρχουν στο Loop και θα μετατρέπεται ο κώδικας όπως παρακάτω:

Πρωτότυπος κώδικας	Προ-μεταλλαγμένο – Φάση 1

Unit_Under_Test(…) {
…
Stat(0)
While (Condition+)
{
Statement*
}
Stat(N+1)
…
}
	LoopFunction(param-list) {
	Stat(0)
	While (Condition+) {
	Statement*
	}
	Stat(N+1)
	}
	Unit_Under_Test(…) {
	….
	LoopFunction(param-list);
	….
	}

Πίνακας 7: Πρώτη φάση μετασχηματισμού επαναληπτικής δήλωσης

Έπειτα θα δημιουργούνται συναρτήσεις μεταλλαγμένες τις οποίες καλεί μια άλλη LoopCallFunction για να εκμεταλλευτούμε τα συμπεράσματα του [OL94] για BB-WEAK/N. Μάλιστα θα μπορούσε να εφαρμοστεί και κάθε τελεστής (operator) που υπάρχει στο Mothra για τα loop σε κάθε μεταλλαγμένη έκδοση αυτής της LoopFunction, όπως παρακάτω:

Προ-μεταλλαγμένο – Φάση 1	Μεταλλαγμένο – Φάση 2
LoopFunction(param-list) {	LoopFunction(param-list) {
Stat(0)	Stat(0)
While (Condition+) {	While (Condition+) {
Stat(1)	Stat(1)
.....	.....
Stat(N)	Stat(N)
}	}
Stat(N+1)	Stat(N+1)
}	}
Unit_Under_Test(…) {	LoopFunction_m1(param-list) {
....	While (Condition+) {
LoopFunction(param-list);	Stat(0)
....	Stat(1)
}	.....
	Stat(N)
	}
	Stat(N+1)
	}
	LoopFunction_m2(param-list) {
	Stat(0)
	Stat(1)
	While (Condition+) {
	Stat(2)
	.....
	Stat(N)
	}
	Stat(N+1)
	}
	LoopFunction_m3(param-list) {
	Stat(0)

```
                While (Condition+)  {
                        Stat(1)
                        .....
                        Stat(N-1)
                }
                Stat(N)
                Stat(N+1)
        }
        LoopFunction_m4(param-list) {
                Stat(0)
                While (Condition+)  {
                        Stat(1)
                        ……
                        Stat(N)
                        Stat(N+1)
                }
        }
        LoopFunction_m5(param-list) {
                Stat(0)
                Condition+ = mutated Condition
                While (Condition+)  {
                        Stat(1)
                        .....
                        Stat(N)
                }
                Stat(N+1)
        }
        LoopFunction_m6(param-list) {
           While (Mutation Loop Condition ){
                Mutated_Stat(0)
                While (Condition+)  {
                        Mutated_Stat(1)
```

```
                    .....
                    Mutated_Stat(N)
                }
                Mutated_Stat(N+1)
            }
        }
        LoopDoMutation() {
        /* Έλεγχος όλων των παραπάνω μεθόδων
        και σύγκριση των τιμών με αυτές κατά
        την εκτέλεση της LoopFunction */
        }
        Unit_Under_Test(…) {
        ....
        LoopDoMutation (…)
        ....
        }
```

Πίνακας 8: Δυνατές μεταλλάξεις επαναληπτικής δήλωσης

Η έκτη εκδοχή του LoopFunction δεν είναι απαραίτητα μια που ελέγχει με βάση την αδύναμη μετάλλαξη κάθε έκφραση στο κύκλο. Θα δημιουργείται μια μέθοδος για κάθε μετάλλαξη, ώστε να ελεγχθεί το αποτέλεσμα του κύκλου σε κάθε περίπτωση ύπαρξης λάθους σε κάποια εσωτερική σε αυτόν έκφραση. Έτσι θα δημιουργούνται δηλαδή από τον μετασχηματιστή επιπλέον αντίστοιχες μέθοδοι, όσες το πλήθος των δυνατών λαθών, που μπορούν να φυτευτούν μέσα στο κύκλο.

Επίσης είναι φανερό ότι μετάλλαξη ανάθεσης τιμής σε μεταβλητή μπορεί και πρέπει να γίνει στις περιπτώσεις που αυτή είναι μέρος ενός κύκλου. Αν στο πρωτότυπο πρόγραμμα κάποια μεταβλητή λαμβάνει την τιμή κάποιας άλλης θα πρέπει να αντικατασταθεί η τελευταία με την αρνητική της τιμή και όλες τις λοιπές δυνατές περιπτώσεις. Το ίδιο θα πρέπει να γίνει ακόμα και στην περίπτωση ανάθεσης σταθερής τιμής, όπου εκεί θα πρέπει να αυξομειωθεί η σταθερά, όπως και να ανατεθεί η αρνητική της τιμή.

6.8. Μετάλλαξη των switch-case

Σύμφωνα με τα συμπεράσματα από το σύστημα για την Ada83 [VM98], δεν θα πρέπει να μεταλλάσσεται ποτέ η μεταβλητή της οποίας την τιμή ελέγχει το switch. Στο παράδειγμα που ακολουθεί δεν θα πρέπει να αλλάξουμε ποτέ το I. Αυτό που μπορεί να αλλάξει είναι μόνο οι τιμές στα case και ο κώδικας που εκτελείται σε κάθε περίπτωση. Το πρώτο αν γίνεται με ανάλογο τρόπο με την μετάλλαξη των σταθερών τιμών θα οδηγούσε στο παρακάτω παράδειγμα:

Πρωτότυπος κώδικας	Μεταλλαγμένη έκδοση

switch(I) {
case 1:
case 2:
default:
}
	switch(I) {
	case 2:
	case 2:
	default:
	}

Πίνακας 9: Θεωρητική μετάλλαξη switch-case δήλωσης

Κάτι τέτοιο όμως δεν επιτρέπεται από το compiler της JAVA. Άρα αυτό που μπορεί να γίνει είναι να αφαιρούνται ολόκληρες περιπτώσεις αναγκάζοντας την εκτέλεση της default. Σε περίπτωση που υπάρχει ήδη η επιλογή "default" θα μπορούσε να αλλάξει η τιμή στο case με κάποια μικρότερη της ελάχιστης περίπτωσης ή με την αμέσως μεγαλύτερη της μέγιστης. Επίσης θα πρέπει να δημιουργηθεί μετάλλαξη αφαιρώντας και την default περίπτωση. Αν αυτή πάλι απουσιάζει (κάτι το οποίο δεν προτείνεται στους προγραμματιστές, αλλά και δεν απαγορεύεται από τους μεταφραστές), τότε αρκεί να αντικατασταθεί όλη η case από την default περίπτωση που δεν κάνει τίποτα. Για παράδειγμα:

Πρωτότυπος κώδικας	Μεταλλαγμένη έκδοση

switch(I) {
case 1:
case 2:
}
	switch(I) {
	case 3:
	default:
	}

Πίνακας 10: Μετάλλαξη switch-case δήλωσης αν δεν υπάρχει default περίπτωση

Πρωτότυπος κώδικας	1η Μετάλλαξη	2η Μετάλλαξη

switch(I) {
case 1:
case 2:
default:
}
	switch(I) {
	case 0:
	case 2:
	default:
	}
		switch(I) {
		case 3:
		case 2:
		default:
		}

Πίνακας 11: Μεταλλάξεις στην περίπτωση που υπάρχει default περίπτωση

Τα παραπάνω θα γίνονται με τρόπο ανάλογο με αυτό που περιγράφηκε σε προηγούμενο σημείο της εργασίας για την μετάλλαξη των κυκλικών δηλώσεων (iterative statements). Θα δημιουργούνται από το μετασχηματιστή δηλαδή ξεχωριστές συναρτήσεις που θα περιέχουν μεταλλαγμένη όλη την switch-case δήλωση. Οι μεταλλάξεις αυτές θα περιλαμβάνουν την διαγραφή των case, όπως και τις μεταλλάξεις για κάθε δήλωση που περιέχεται σε αυτές.

6.9. Μείωση πλήθους μεταλλάξεων

Η προσέγγιση που προτείνεται από την παρούσα εργασία μειώνει το πλήθος των εκτελούμενων μεταλλάξεων. Αυτό γίνεται γιατί κάθε φορά ανάλογα με την ροή της εκτέλεσης του προγράμματος μετρώνται και εκτελούνται οι όποιες μεταλλάξεις. Αυτό δεν γινόταν σε προηγούμενα συστήματα μετάλλαξης όπως το Mothra, που ανάγκαζαν την εκτέλεση όλων των κλάδων έστω μια φορά. Αυτό οδηγούσε σε εκτέλεση μονοπατιών εκδόσεων που ίσως στην πραγματικότητα να ήταν ανέφικτο να βρεθούν δεδομένα για να τα καλύψουν.

Αυτό που έχει ενδιαφέρον είναι ο υπολογισμός κάποιας εκτίμησης του πλήθους των μεταλλάξεων για κάθε πρόγραμμα σύμφωνα με την προτεινόμενη τεχνική. Ο Budd εκτίμησε το πλήθος των μεταλλάξεων για την ισχυρή εκδοχή ως ανάλογο του γινομένου των αναφορών δεδομένων και του μεγέθους του συνόλου των δομών τους [ORZ93]. Η εκτίμηση του A.T. Acree στην διδακτορική του διατριβή [Acr80] ανέφερε N^2 μεταλλάξεις για ένα πρόγραμμα στο οποίο εμφανίζονται N αναφορές σε μεταβλητές. Στην προτεινόμενη προσέγγιση το μέγιστο αναμενόμενο πλήθος των μεταλλάξεων εκτιμάται από τον τύπο:

$$\sum_{t=1}^{T}(VR_t\ (V_t-1))+\sum_{l=0}^{L}\left\lceil\sum_{t=1}^{T}(IV_t\cdot V_t)+3\cdot IC\right\rceil_l+8\cdot L+\sum_{s=0}^{S}(2\cdot C_s+1)+$$
$$3\cdot CR+4\cdot AOR+2\cdot LOR+2\cdot BOR+5\cdot ROR$$

Μεταβλητή	Εξήγηση
VR_t	Αναφορές σε μεταβλητές τύπου t
V_t	Μεταβλητές τύπου t ή τύπου επιτρεπτής μετάπτωσης σε t (type casting)
T	Διαφορετικοί τύποι μεταβλητών στο πρόγραμμα.
IV_t	Πλήθος αρχικοποιήσεων με μεταβλητές τύπου t.
IC	Πλήθος αρχικοποιήσεων με σταθερές τιμές.
L	Πλήθος επαναληπτικών δηλώσεων (while, for, do – while)
S	Πλήθος switch δηλώσεων
C_s	Πλήθος case περιπτώσεων σε κάθε switch-case δήλωση s
CR	Πλήθος εμφανιζόμενων σταθερών μεταβλητών ή τιμών.
AOR	Πλήθος εμφανιζόμενων αριθμητικών τελεστών.
LOR	Πλήθος εμφανιζόμενων λογικών τελεστών.
BOR	Πλήθος εμφανιζόμενων δυαδικών (binary) τελεστών.
ROR	Πλήθος εμφανιζόμενων σχεσιακών τελεστών.

Πίνακας 12: Επεξήγηση συμβόλων τύπου εκτίμησης μέγιστου πλήθους μεταλλάξεων σύμφωνα με το προτεινόμενο σύστημα

Βέβαια θα μπορούσε να τροποποιηθεί με τρόπο, ώστε να εκτελεί επιλεκτική μετάλλαξη. Για να γίνει αυτό θα πρέπει να ληφθούν υπόψη οι τελεστές που αναφέρονται σε αντίστοιχη υλοποίηση από τα συστήματα για τις γλώσσες Fortran-77 (Mothra [DGK+88, KO91, OC94, VM98]) και Ada83 [VM98]. Οι τελεστές που θα παραμείνουν προς χρήση σε αυτή την περίπτωση είναι οι παρακάτω:

- εισαγωγής απόλυτης τιμής (με τον τρόπο που εδώ προτάθηκε),
- αντικατάστασης αριθμητικών,
- λογικών,
- σχεσιακών και
- unary τελεστών,
- αντικατάστασης καλούμενου υποπρογράμματος.

Μάλιστα στην περίπτωση που θα πρέπει να εκτελεστεί επιλεκτική μετάλλαξη ο μετασχηματισμός του προγράμματος που ελέγχεται θα γίνεται χρησιμοποιώντας διαφορετικές εκδόσεις των σχηματικών συναρτήσεων. Έτσι η ταχύτητα του εκτελούμενου υπέρ-μεταλλαγμένου προγράμματος θα είναι βελτιστοποιημένη, αφού θα απουσιάζουν περιττοί έλεγχοι για την εφαρμογή ή όχι των «περιττών» τελεστών μετάλλαξης.

Βέβαια κάτι τέτοιο δεν είναι σίγουρο κατά πόσο έχει αξία, γιατί η επιλεκτική προσέγγιση στη μέθοδο των μεταλλάξεων προτάθηκε για να λύσει προβλήματα της ισχυρής εκδοχής. Η αδύναμη μετάλλαξη αποτελεί ήδη ένα υποσύνολο της μεθόδου και τα αποτελέσματα αξιολόγησης δεδομένων που δίνει είναι σχετικά υπερτιμημένα με αυτά της ισχυρής. Επιπλέον το παρόν προτεινόμενο σύστημα δεν φαίνεται να απέχει ιδιαίτερα από την επιλεκτική μετάλλαξη. Αυτό οφείλεται στις παραδοχές που έγιναν στην διάρκεια της ανάλυσης του και στο ότι αρκετοί τελεστές δεν επιλέχθηκαν διότι θα προκαλούσαν αλλαγή ροής προγράμματος και κατά συνέπεια διαφορετικό αποτέλεσμα στο «επόμενο βήμα».

6.10. Εύρεση ισοδύναμων μεταλλάξεων

Όπως αναφέρθηκε προηγουμένως η δημιουργία των μεταλλάξεων γίνεται ψευδο-δυναμικά και εξαρτάται από την δυνατότητα των δεδομένων να καλύψουν τα μονοπάτια που τις περιέχουν. Η προσέγγιση λοιπόν αυτή δεν δημιουργεί εκδόσεις με μη προσβάσιμα μονοπάτια, τα οποία θα σήμαιναν αμέσως ισοδυναμία. Βέβαια ακόμα δεν μπορεί να κριθεί μέσα από το σχήμα αν για παράδειγμα μια μετάλλαξη αντικατάστασης μεταβλητής δημιουργεί συνολικά μια ισοδύναμη ή μη έκδοση. Αν μέσα στο σχήμα υπάρχει ο κώδικας I = K και το αποτέλεσμα είναι ίδιο με τον μεταλλαγμένο I = X, δεν είναι γνωστό αν είναι ισοδύναμη έκδοση, έχοντας υπόψη μόνο αυτή την πληροφορία.

Πρωτότυπος κώδικας 1	Ισοδύναμη μετάλλαξη	Πρωτότυπος κώδικας 2	Ισοδύναμη μετάλλαξη
K = X; If (a < 0) I = K;	K = X; If (a < 0) I = X;	K = b; X = b; If (a < 0) I = K;	K = b; X = b; If (a < 0) I = X;

Πίνακας 13: Περιπτώσεις από όπου μπορεί να προκύψουν ισοδύναμες εκδόσεις

Για να αντιμετωπιστεί το παραπάνω είναι δυνατό να χρησιμοποιούνται οι κλήσεις καταχώρησης των μεταβλητών που θα προστίθενται από το μετασχηματιστή που θα υλοποιηθεί με διαφοροποιημένο τρόπο. Θα μπορούσαν να κρατούν το ιστορικό ανάθεσης της κάθε τιμής στις μεταβλητές. Έτσι αν έφτανε ο έλεγχος στην μετάλλαξη της πρώτης ή της δεύτερης περίπτωσης θα ήταν δυνατό να την κρίνει ως ισοδύναμη βλέποντας ότι η τιμή που ανατέθηκε στο K είναι αυτή του X ή επειδή και στις δύο μεταβλητές θα είχε ανατεθεί τελευταία φορά η ίδια μεταβλητή ή σταθερά. Αυτός ο έλεγχος μπορεί να γίνεται αναδρομικά προς τα πίσω, αν δηλαδή στην τρίτη περίπτωση είχε ανατεθεί ίδια μεταβλητή ή σταθερά στις b και c πριν την χρήση τους. Επίσης σε αυτή την περίπτωση αν οι τιμές των K,X ήταν ίδιες αλλά προέρχονταν από διαφορετικές παραστάσεις και όχι από απλές αναφορές σε μεταβλητές, τότε θα ήταν ξεκάθαρη η μη ισοδυναμία τους. Αν συνέβαινε κάτι τέτοιο θα σήμαινε αδυναμία των δεδομένων να σκοτώσουν τις μεταλλάξεις.

6.11. Παράδειγμα από το πρότυπο

Στα πλαίσια της παρούσας εργασίας και στην προσπάθεια καθορισμού των προδιαγραφών ενός συστήματος, που θα εφαρμόζει τα όσα έχουν περιγραφεί παραπάνω, δημιουργήθηκε ένα πρότυπο. Αυτό το αποτελούν κλάσεις που υλοποιούν τα στατικά σχήματα και απαραίτητες συναρτήσεις παρακολούθησης στοιχείων του προγράμματος, χρήσιμων για την εφαρμογή κάποιων μεταλλάξεων. Επίσης, υπάρχει και μια κλάση που χρησιμεύει στην δημιουργία του κώδικα για κάθε μεταλλαγμένη έκδοση. Με την βοήθεια αυτού του προτύπου δοκιμάστηκε η εμφύτευση κλήσεων σε συναρτήσεις με τρόπο ανάλογο με αυτό που θα έπρεπε να λειτουργεί το ζητούμενο σύστημα. Για παράδειγμα για το πρόγραμμα Newton το οποίο παρατίθεται στο σχετικό παράρτημα, δημιουργήσαμε το υπερ-μεταλλαγμένο πρόγραμμα Newton_mut, όπως ακολουθεί:

```
/* import που υπήρχαν ήδη στο πρόγραμμα */
import java.lang.*; // System
import java.io.*;  // println

/* import που πρόσθεσε ο μετασχηματιστής */
/* package με συναρτήσεις που δημιουργήθηκαν για την μετάλλαξη */
import JMutOps.*;
/* package που περιέχει αρχεία με τα μεταλλαγμένα loop */
import JMSTests.JMSLoopSchemas.*;

public class NewtonWeakMut {
    private JavaMutantSchema jms;

public NewtonWeakMut (JavaMutantSchema jmsT) { jms = jmsT; }

public double Newton_Loop_mut1(double num) {
  double res, NewGuess, d, e;
  jms.trace.AddVar(0, "num", num);
  jms.trace.AddVar(0, jms.JMS_DBL, "res");
  jms.trace.AddVar(0, jms.JMS_DBL, "NewGuess");
  jms.trace.AddVar(0, jms.JMS_DBL, "d");
  jms.trace.AddVar(0, jms.JMS_DBL, "e");
  int lineno;                    // added code for mutation

  jms.AddTrace(0, "public double Newton(double num) {\n" +
                  "double res, NewGuess, d, e;");
    e = 0.001;
  jms.trace.SetValue("e", 0.001);
  jms.AddTrace(1, "e = 0.001;");
  NewGuess = (num / 2.0) + 1.0;
  jms.AddTrace(2, "NewGuess = (num / 2.0) + 1.0;");


  NewGuess = jms.ArithWeakOp(                    // added code
             jms.ArithWeakOp("num",             // added code
             2, jms.ari.AO_DIV,"NewGuess = (num ", " 2.0) + 1.0;"),
// added code             1.0, 2, jms.ari.AO_ADD,"NewGuess = (num /
2.0) ", " 1.0;"); // added code
  jms.trace.SetValue("NewGuess", NewGuess);
  res = 0.0;
  jms.AddTrace(3, "res = 0.0;");
  jms.trace.SetValue("res", 0.0);

  /* d = NewGuess - res; -->
   * Είναι η Statement(0) που μετατοπίστηκε μέσα στη μετάλλαξη του
loop
   */

  Newton_JMSLoop jmsloop = new Newton_JMSLoop();
  res = jmsloop.Newton_LoopCall1_DoMutation(num,
res, NewGuess, e, 4, 12, jms);
  jms.AddTrace("return res;");
  jms.AddTrace("}");
 return res; }}
```

Με μια πρώτη ματιά το παραπάνω φαίνεται ότι είναι αρκετά πιο περίπλοκο από το πρωτότυπο πρόγραμμα. Αυτό συμβαίνει γιατί έγινε προσπάθεια να διατηρηθούν πληροφορίες από την φάση της μετάφρασης και στην φάση της εκτέλεσης του προγράμματος και να διευκολυνθεί η αποτελεσματικότερη δημιουργία μεταλλάξεων. Η βασική κλάση είναι η JavaMutantSchema στην οποία θα φυλάσσονται όλες οι αναγκαίες πληροφορίες για την δημιουργία και εκτέλεση των μεταλλάξεων, αλλά και για την προς τα πίσω δημιουργία των μεταλλαγμένων εκδόσεων (decompilation). Φυλάσσονται στοιχεία για τις μεταβλητές που χρησιμοποιούνται, όπως τα ονόματα, οι τύποι τους και οι τιμές τους μετά από κάθε αλλαγή. Αυτό συγκεκριμένα γίνεται από την κλάση JMSTrace, η οποία αποτελεί μέρος του προτύπου.

Σημαντικό είναι να σημειωθεί ότι δεν έχει μεταλλαχθεί η ανάθεση σταθεράς σε μεταβλητή, σύμφωνα με τα όσα έχουν περιγραφεί στην ανάλυση του συστήματος. Επίσης έχει παραμείνει ο πρωτότυπος κώδικας για να αποτελεί οδηγό, στην ουσία όμως αυτό δεν είναι αναγκαίο, γιατί όλα τα σχήματα επιστρέφουν την ίδια τιμή που θα προέκυπτε από την εκτέλεση του πρωτότυπου και όχι του μεταλλαγμένου κώδικα. Ένα σχήμα για παράδειγμα εφαρμόζεται με την κλήση της ArithWeakOp, η υλοποίηση της οποίας δίνεται στο παράρτημα Β. Συγκεκριμένα οι παράμετροι αυτής της μεθόδου είναι οι δύο τελεσταίοι που παίρνουν μέρος στην πράξη, ο κωδικός της πράξης, και πληροφορίες για την τρέχουσα γραμμή και τα αλφαριθμητικά, αν θεωρούσαμε τον κώδικα απλό κείμενο. Οι τελευταίες πληροφορίες είναι διαθέσιμες στην φάση της ανάλυσης του κώδικα και θα μπορεί ο ζητούμενος μετασχηματιστής να τις χρησιμοποιήσει.

Ένα επίσης σημαντικό σχήμα υλοποιείται με την μέθοδο Newton_LoopCall1_DoMutation. Αυτό είναι δυναμικά σχηματιζόμενο και εξαρτάται από το υπό έλεγχο πρόγραμμα. Περιέχει κλήσεις σε όλες τις μεθόδους που θα δημιουργεί ο μετασχηματιστής και οι οποίες περιλαμβάνουν τον κώδικα του πρωτότυπου προγράμματος με ένα μόνο λάθος. Όλες είναι μέλη μιας δυναμικά δημιουργημένης κλάσης, που εδώ ονομάζεται Newton_JMSLoop. Μετά από κάθε τέτοια μέθοδο και ενώ ακόμα εκτελείται η Newton_LoopCall1_DoMutation (βλ. Παράρτημα Β) ελέγχεται η κατάσταση της κλάσης και συγκρίνεται μέρος της μνήμης του προγράμματος και όχι μια μόνο μεταβλητή.

Κεφάλαιο 7ο: Αξιολόγηση καινούριας προσέγγισης

7.1. Θεωρητικά

Σύμφωνα με τα συμπεράσματα από την μελέτη εφαρμογής σχημάτων των Untch, Offutt και Harrold [UOH93] το σύστημα που προτείνεται, επειδή βασίζεται σε σχήματα στο επίπεδο της γλώσσας γραφής του προγράμματος, είναι **γρηγορότερο** από αυτά που βασίζονται σε διερμηνέα, όπως το Mothra. Ένα MSG (Mutant Schema Generator) σύστημα κατορθώνει την βέλτιστη εκτέλεση των μεταλλάξεων χωρίς να χρειάζεται να τους φυλάει σε διαφορετικό μέρος. Παρόλο που πρέπει να εκτελεστούν αρκετές μετά-συναρτήσεις το πρόγραμμα εκτελείται μεταφρασμένο **σε ταχύτητες επιπέδου γλώσσας μηχανής**. Επίσης είναι αρκετά **πιο οικονομική η σχεδίαση και υλοποίηση** ενός τέτοιου συστήματος από αντίστοιχα βασισμένα σε διερμηνέα. Κάτι τέτοιο τονίζεται και από τον Voas [VM98]. Το μεταλλαγμένο πρόγραμμα είναι στην ίδια γλώσσα με αυτή του πρωτότυπου. Για αυτό μπορεί να χρησιμοποιηθεί ο **ίδιος μεταφραστής** και το **περιβάλλον ελέγχου** να είναι ίδιο με αυτό της πραγματικής χρήσης, ενώ διατηρούνται οι λειτουργίες και η συμπεριφορά του. Ένα σύστημα, που επεμβαίνει στο επίπεδο της γλώσσας γραφής του προγράμματος, επιτρέπει την εκτέλεση του ελέγχου **σε διαφορετικές πλατφόρμες και περιβάλλοντα**, αφού αρκεί η επαναμετάφραση του κώδικα. Αυτή η ανεξαρτησία σχετικά με την πλατφόρμα εκτέλεσης καθιστά ευκολότερη την **υλοποίηση κατανεμημένου ελέγχου** (heterogeneous distributed computing). Τέλος είναι δυνατό **να υλοποιηθεί μερικώς** το σύστημα (όλα τα σχήματα ή η πλήρης λειτουργικότητα τους) και παρόλα αυτά να εφαρμόζεται ο έλεγχος με την μέθοδο των μεταλλάξεων. Σε αντίθεση τα συστήματα που βασίζονται σε διερμηνέα πρέπει να περιμένουν την υλοποίηση όλου του συστήματος πριν ξεκινήσει η χρήση του. Επιπλέον η χρήση σχημάτων επιτρέπει την **κλιμάκωση** και βελτίωση του συστήματος με τρόπο αρκετά ευέλικτο.

Σύμφωνα με τους Untch, Offutt και Harrold ένα σύστημα MSG (Mutant Schema Generator) αν και ταχύτερο από συστήματα που βασίζονται σε διερμηνέα είναι αργότερο από (compiler-integrated) συστήματα ενσωματωμένα σε μεταφραστή. Βέβαια υποστηρίζουν ότι η μεταφερσιμότητα του συστήματος και η ευκολία υλοποίησης του είναι αρκετά συμφέρουσα ανταλλαγή σε σχέση με την ταχύτητα που θα επιτυγχανόταν με την υλοποίηση ενός συστήματος στο επίπεδο μεταφραστή.

Η καινούρια προσέγγιση υπακούοντας στις αρχές της «αδύναμης» μετάλλαξης απαιτεί **λιγότερη εκτέλεση κώδικα**. Αρκεί η εύρεση διαφορετικών αποτελεσμάτων σε επίπεδο συστατικού προγράμματος και όχι στο σύνολο της εκτέλεσης του κώδικα. Η αποφυγή εκτέλεσης κώδικα πριν και μετά το σημείο μετάλλαξης όμως γίνεται εδώ με διαφορετικό τρόπο από αυτό που χρησιμοποιήθηκε με το σύστημα Leonardo [OL94] από τους Offutt και Lee για την αξιολόγηση της «αδύναμης» μετάλλαξης. Έστω ότι το πρόγραμμα απαιτεί χρόνο εκτέλεσης Τ. Για περιπτώσεις που θα είχαμε έστω Ν πιθανές μεταλλάξεις ενός συγκεκριμένου σημείου του προγράμματος με την μέθοδο της ισχυρής μετάλλαξης θα χρειαζόταν χρόνος (Τ*Ν). Τώρα ο χρόνος που χρειάζεται για την εκτέλεση με τα ίδια δεδομένα όλων αυτών είναι (Ν+Τ) = Τ. Με το σύστημα Leonardo πάλι θα χρειαζόμασταν περισσότερο χρόνο, αφού εκεί κάποιες μεταλλάξεις θα σταματούσαν πριν εκτελεστούν μέχρι τέλους, αλλά παρόλα αυτά θα χρειαζόταν η εκτέλεση του ίδιου κώδικα πολλές φορές για να φτάσει ο έλεγχος στο επιθυμητό σημείο μετάλλαξης σε άλλες περιπτώσεις.

Επιτυγχάνεται αποφυγή μετάφρασης κάθε έκδοσης. Τώρα υπάρχει **μια έκδοση μόνο**, οπότε εξοικονομείται ο χρόνος που χρειαζόταν για την μεταγλώττιση όλων αυτών των εκδόσεων. Βέβαια χωρίς να υπάρχει ένα σύστημα που να δημιουργεί αυτή την υπερ-μεταλλαγμένη έκδοση δεν είναι εύκολη η σύγκριση των αναγκαίων χρόνων. Παρόλα αυτά όμως εκτιμάται ότι ακόμα κι αν δεκαπλασιαστεί ο κώδικας, η μεταγλώττιση του δεν θα απαιτεί χρόνο όσο την μεταγλώττιση του συνολικού πλήθους των μεταλλάξεων. Για παράδειγμα, έστω ότι το πρόγραμμα που παραπάνω αναφέρθηκε ο έλεγχος του από το Mothra. Εκεί δημιουργήθηκαν 951 μεταλλαγμένες εκδόσεις για πρόγραμμα (ταξινόμησης τριγώνων) 30 γραμμών. Πιστεύεται λοιπόν ότι με την προσέγγιση που προτείνεται από την παρούσα εργασία απαιτείται χρόνος μεταγλώττισης σημαντικά μικρότερος από το χρόνο μεταγλώττισης του εν λόγω προγράμματος επί το παραπάνω πλήθος.

Η «αδύναμη» μετάλλαξη είναι γνωστό ότι δεν παρέχει την ίδια αξιοπιστία για τα δεδομένα ελέγχου και τους δίνει υπερτιμημένο βαθμό MS (Mutation Score) [DO91]. Παρόλα αυτά η προτεινόμενη εκδοχή εκμεταλλευόμενη τα συμπεράσματα των Offutt και Lee [OL94] προσπαθεί να μειώσει αυτή την υπερτίμηση. Αυτό που προτείνεται εδώ είναι η χρήση firm μετάλλαξης στην περίπτωση των κύκλων και των δηλώσεων switch-case. Επιπλέον η χρήση των περισσοτέρων τελεστών μετάλλαξης,

που εφαρμόζονται σε συστήματα ισχυρής μετάλλαξης, σε συνδυασμό με το μετασχηματισμό κύκλων πιστεύεται ότι βοηθάει σε καλύτερες εκτιμήσεις.

Η νέα προσέγγιση εφαρμόζει **αυτοματοποιημένη διαδικασία ελέγχου των αποτελεσμάτων** κάθε μεταλλαγμένου προγράμματος σε κάθε κόμβο. Αυτό το μέρος της διαδικασίας ήταν από τα πιο δαπανηρά και απαιτούσε ανθρώπινη επίβλεψη. Με αυτή την υλοποίηση μπορεί να γίνει με τρόπο που ο ανθρώπινος παρατηρητής θα ελέγχει μόνο τα αποτελέσματα του ελέγχου και όχι κάθε έκδοσης. Για αυτό θα μπορούσαμε να χαρακτηρίσουμε την προσέγγιση ως εμφυτευμένο σε σχήματα έλεγχο με την χρήση μεταλλάξεων (Mutation Testing Injected in Schemas – MuTIS).

Μάλιστα θα μπορούσε να χρησιμοποιηθεί **σε πραγματικές συνθήκες λειτουργικού ελέγχου**, αν οι χρόνοι είναι αποδεκτοί, με σκοπό την αξιολόγηση των δεδομένων ελέγχου και την αρχειοθέτηση τους για μελλοντικούς ελέγχους μετά από βελτίωση ή διόρθωση του προγράμματος. Αυτό που συχνά γίνεται στην διάρκεια του λειτουργικού ελέγχου είναι ότι κάποιος ελεγκτής χρησιμοποιεί δεδομένα για την εξέταση της ομαλής λειτουργίας του προγράμματος και ελέγχει τα τελικά αποτελέσματα. Η προσέγγιση που προτείνεται του δίνει την δυνατότητα να εκτελεί το υπέρ-μεταλλαγμένο πρόγραμμα και με σχετικά εύκολο τρόπο στο τέλος των ελέγχων να έχει πραγματοποιήσει και αξιολόγηση της δύναμης των δεδομένων που χρησιμοποιήθηκαν. Έτσι την επόμενη φορά θα γνωρίζει τα ποσοστά κάλυψης των δεδομένων όπως και την δύναμη να αποκαλύψουν διαφορές («λάθη») στο πρόγραμμα. Σε περίπτωση που αυτός ο έλεγχος μπορεί να γίνει αυτοματοποιημένος τα δεδομένα θα εισάγονται στο πρόγραμμα με βάση αυτή την αξιολόγηση. Αυτό θα βοηθήσει τον ελεγκτή στην απόφαση του για το πότε να σταματήσει τον έλεγχο. Είναι χρήσιμο λοιπόν το σύστημα μετάλλαξης να φυλάει στο τέλος κάθε σεναρίου ελέγχου τα δεδομένα, το χρόνο εκτέλεσης, το πλήθος των δημιουργημένων, «σκοτωμένων» και όποιων ισοδύναμων μεταλλάξεων, όπως και το ποσοστό κάλυψης μονοπατιών. Το τελευταίο αν και έχει σχέση με το πλήθος των δημιουργημένων μεταλλάξεων δεν θα μπορούσε να προκύψει μόνο με αυτή την πληροφορία.

Η προτεινόμενη προσέγγιση αν και δεν λύνει ολοκληρωτικά το πρόβλημα της εύρεσης ισοδύναμων μεταλλάξεων, **μειώνει σημαντικά το πλήθος αυτών που είναι πιθανά ισοδύναμες**. Αυτό όπως εξηγήθηκε και πιο πάνω γίνεται γιατί ο εμφυτευμένος έλεγχος λειτουργεί εκμεταλλευόμενος την κάλυψη των μονοπατιών. Έτσι δεν χρειάζεται ο έλεγχος μεταλλαγμένων εκδόσεων που οφείλονται σε

απροσπέλαστα μονοπάτια, γιατί απλούστατα το πρόγραμμα δεν έφτασε ποτέ εκεί για να τα δημιουργήσει. Έμμεσα δηλαδή εκτελείται και **ανάλυση κάλυψης μονοπατιών** πριν την δημιουργία των μεταλλάξεων. Αυτό σημαίνει ότι δεν χρειάζεται απραίτητα λογισμικό-πράκτορας που θα παρακολουθεί τον έλεγχο, ιδιαίτερα αν τα δεδομένα που χρησιμοποιούνται είναι σίγουρο ότι μπορούν να καλύψουν όλα τα προσπελάσιμα μονοπάτια. Σημαντικό είναι και το πλεονέκτημα ότι **επιταχύνεται η διαδικασία εξόντωσης των "killable mutants"** (που πρέπει να σκοτωθούν).

Η νέα προσέγγιση φαίνεται σα να μην εκμεταλλεύεται όσες τεχνικές έχουν δημιουργηθεί μέχρι τώρα λόγω της διαφορετικής προσέγγισης. Ισχύει άραγε αυτό ή μπορεί να εμπλουτιστεί η προσέγγιση αυτή από ήδη δοκιμασμένες τεχνικές για την ακόμα περαιτέρω επιτάχυνση της διαδικασίας; Στην ουσία όμως μάλλον δεν χάνονται προηγούμενες τεχνικές, αρκεί να βρεθεί τρόπος να εμπλουτιστεί η νέα προσέγγιση με αυτές. Εξάλλου ήδη έχουν ληφθεί σοβαρά υπόψη κάποιες μελέτες (**σχήματα μετάλλαξης, χρήση πληροφορίας μετάφρασης, ανάγκη για ανάλυση κάλυψης μονοπατιών, είδη τελεστών μετάλλαξης, αξιολόγηση αδύναμης μετάλλαξης, firm μετάλλαξη**), ενώ οι υπόλοιπες δεν εφαρμόστηκαν μετά από αξιολόγηση τους. Μάλιστα η νέα προσέγγιση εκμεταλλευόμενη την δυνατότητα των σχημάτων να είναι επεκτάσιμα και κλιμακούμενα (scalable) με τρόπο ευέλικτο, συγχρόνως επεκτείνει αυτή την ευελιξία αναθέτοντας τον έλεγχο σε κλάσεις οι οποίες μπορούν να τροποποιηθούν προσθέτοντας ή αφαιρώντας ελέγχους.

Συγκεντρώνοντας όλα τα παραπάνω η συγκεκριμένη προσέγγιση πετυχαίνει:

- Εκτέλεση του ελάχιστου απαραίτητου κώδικα
- Μετάφραση ενός προγράμματος μόνο
- Εκμετάλλευση συμπερασμάτων αξιολόγησης αδύναμης μετάλλαξης
- Εκμετάλλευση εγκυρότητας μεταφραστή γλώσσας
- Χρήση διαφόρων προσεγγίσεων:
  - Έμμεση ανάλυση ροής δεδομένων και κάλυψης μονοπατιών
  - Χρήση σχημάτων μετάλλαξης
  - Χρήση μεγάλου εύρους τελεστών μετάλλαξης
  - Μερική εφαρμογή firm mutation
  - Εκμετάλλευση πληροφορίας κατά την μετάφραση
- Αυτοματοποιημένος run-time έλεγχος αποτελεσμάτων
- Η εκτέλεση των μεταλλάξεων οδηγείται από τα δεδομένα

- Δυναμική δημιουργία μεταλλάξεων
- Αποφυγή δημιουργίας μεταλλάξεων από νεκρό ή απροσπέλαστο κώδικα
- Μείωση πλήθους πιθανά ισοδύναμων και του συνόλου των μεταλλάξεων
- Επεκτασιμότητα ελέγχου
- Βελτιστοποίηση κώδικα από τον μεταφραστή της γλώσσας
- Επιτάχυνση εκτέλεσης των μεταλλάξεων
- Αξιολόγηση δεδομένων ελέγχου στην διάρκεια λειτουργικού ελέγχου
- Δυνατότητα συνδυασμού λειτουργικού ελέγχου και μεθόδου μεταλλάξεων
- Έλεγχος σε πραγματικές συνθήκες χρήσης του προγράμματος

7.2. Πειραματικά

Παρατηρήθηκε ότι κάθε φορά το πόσες μεταλλάξεις θα εκτελεστούν εξαρτάται από τα δεδομένα και το ποσοστό κάλυψης κόμβων του προγράμματος. Σημαντικό εδώ είναι ότι με την εμφύτευση του ελέγχου των δυνατών μεταλλάξεων δεν χρειάζεται να ελεγχθεί η κάλυψη των κλάδων, αφού τα δεδομένα οδηγούν την εκτέλεση του προγράμματος και έτσι είναι σα να γίνεται ταυτόχρονα ανάλυση data flow και path coverage. Αυτό πιστεύω ότι σε προηγούμενες υλοποιήσεις χρειαζόταν να γίνεται από το διερμηνέα κάθε φορά για να αποφασίζει ποιες μεταλλάξεις να δοκιμάσει. Ειδικά στο strong mutation θα έπρεπε να εκτελεστούν όλες οι δημιουργημένες μεταλλαγμένες εκδόσεις αν δεν ελεγχόταν το κατά πόσο ένα σημείο είναι προσπελάσιμο (feasible) ή όχι. Αυτό ίσως μπορεί να χρησιμοποιηθεί ως επιχείρημα για την ενίσχυση της πιστότητας των δεδομένων ελέγχου. Ειδικά το γεγονός ότι κάποιοι κλάδοι δεν θα εκτελούνται λόγω μη τήρησης συνθηκών είναι σημαντικό ακόμα και για την μείωση του κόστους της μεθόδου, γιατί μπορεί να αρκεστεί ακόμα και στον έλεγχο των μισών μεταλλάξεων (μείωση υπολογιστικού κόστους εκτέλεσης μέχρι το μισό).

Επειδή αξιολογούνται και τα δεδομένα μαζί θα πρέπει να βρίσκουν όσο το δυνατό περισσότερα λάθη σε λιγότερο χρόνο καλύπτοντας το μέγιστο δυνατό πλήθος κόμβων και κλάδων. Άρα είναι χρήσιμο να υπολογίζεται το κλάσμα του βαθμού αξιοπιστίας δεδομένων ελέγχου ή σκορ μετάλλαξης (Mutation Score - MS)

Αν στο Statement(n) παρατηρηθεί χρήση μεταβλητής που έχει τεθεί σε προηγούμενο statement σε «ουδέτερη» τιμή, τότε ίσως να υπήρχε όφελος αν δεν

μεταλλασσόταν η Statement(n) ως προς την πράξη της με την συγκεκριμένη μεταβλητή. Για παράδειγμα:

```
Res = 0;
N = A + Res;
```

Αν τώρα μεταλλάξουμε την δεύτερη γραμμή δεν έχει νόημα να δοκιμαστεί η κλήση:

```
N = A – Res;
```

Στην ουσία μια τέτοια μετάλλαξη (στην προτεινόμενη προσέγγιση) δεν συνεπάγεται ιδιαίτερο υπολογιστικό φόρτο (είναι εκτέλεση μιας πράξης, ενός ελέγχου και μιας καταχώρησης του είδους της μετάλλαξης και όχι επανεκτέλεση ολόκληρου προγράμματος ή μέρους του). Κατά συνέπεια ίσως τελικά κάτι τέτοιο να μην χρειάζεται εκτός κι αν μπορεί να έχει κέρδος σε μεγάλες μονάδες προγραμμάτων υπό έλεγχο. Αντιθέτως οι απαραίτητοι έλεγχοι για την αποφυγή τελικά της εκτέλεσης αυτής της περίπτωσης θα καθυστερούσαν την όλη διαδικασία στο σύνολο της.

7.3. Δοκιμάζοντας προγράμματα

Στην προσπάθεια αξιολόγησης της προτεινόμενης τεχνικής χρησιμοποιήθηκε υπολογιστής Intel Pentium III στα 1000 MHz με μνήμη RAM 128 MB και τρέχοντας σε λειτουργικό Microsoft Windows Me (έκδοση 4.90.3000) την έκδοση JDK 1.4 για JAVA. Επίσης δοκιμάστηκαν, αφού πρώτα μεταφράστηκαν σε JAVA (βλ. παράρτημα Α), τα προγράμματα Newton και TriType όπως βρέθηκαν στην σχετική αρθρογραφία.

Η λογική ελέγχου ήταν να δοκιμαστεί κάθε πρωτότυπο πρόγραμμα με κάποια δεδομένα και έπειτα το υπερ-μεταλλαγμένο πρόγραμμα με τα σχήματα χρησιμοποιώντας ακριβώς τα ίδια δεδομένα κάθε φορά. Σε κάθε έλεγχο υπολογίστηκε ο χρόνος που χρειάστηκε για την ολοκλήρωση της εκτέλεσης του μεταλλαγμένου προγράμματος. Ακόμα αριθμήθηκαν δυναμικά οι μεταλλάξεις που δημιουργήθηκαν (σύμφωνα με τον κλασσικό ορισμό τους, δηλαδή εκδόσεις που περιέχουν ένα μόνο λάθος) και το μέρος αυτών που σκοτώθηκε δίνοντας διαφορετικό αποτέλεσμα. Τέλος υπολογίστηκε ο βαθμός αξιοπιστίας (MS – Mutation Score) των δεδομένων ελέγχου σε σχέση με τη μέθοδο των μεταλλάξεων ως ο λόγος των σκοτωμένων μεταλλάξεων προς το συνολικό τους πλήθος. Αυτό έγινε γιατί δεν

υλοποιήθηκαν έλεγχοι στο επίπεδο του πρωτοτύπου για πιθανή ισοδυναμία μεταξύ δύο εκδόσεων, παρόλο που ο σχεδιασμός έγινε λαμβάνοντας τους υπόψη.

Στον σχήμα 18 φαίνονται τα δεδομένα που χρησιμοποιήθηκαν για δοκιμή, ο χρόνος του πρωτότυπου και ο αντίστοιχος του υπερ-μεταλλαγμένου, το πλήθος των δυναμικά δημιουργημένων μεταλλάξεων και αυτών που σκοτώθηκαν και τέλος η αξιολόγηση των δεδομένων, όπως αυτή δίνεται από τον συντελεστή μετάλλαξης (Mutation Score – MS).

Δεδομένα ελέγχου	Χρόνος Πρωτότυπου	Χρόνος Μεταλλαγμένου	Πλήθος Μεταλλάξεων	Πλήθος Σκοτωμένων	Mutation Score - MS (Συντελεστής Μετάλλαξης)
Για το πρόγραμμα Newton					
-6059.235222941958	3.8409963250160217	1.4147069156169891	44	21	0.4772727272727273
-33.14550107543613	1.4339831396937 37	1.740726463496685	44	21	0.4772727272727273
-11139.915412380062	1.1934494376182556	1.387887828052044	44	21	0.4772727272727273
-15348.456984475855	1.4909736961126328	1.449906975030899	44	21	0.4772727272727273
14983.854509036664	1.4817546382546425	2.026517376303673	104	79	0.7596153846153846
34917.18641192952	4.978293299674988	1.9929935112595558	104	79	0.7596153846153846
9469.800989448835	4.918788447976112	2.0005363821983337	104	79	0.7596153846153846
38387.5014140892	4.9254932180047035	2.0650698095560074	104	79	0.7596153846153846
Για το πρόγραμμα TriType					
2454, 3254, 17000	1.8128027617931366	2.3894131630659103	154	75	0.487012987012987
9576, 8950, 18648	0.8439631909132004	1.9033171832561493	154	75	0.487012987012987
18094, 3001, 17769	1.1238874271512032	13.361772738397121	154	68	0.44155844155844154
5777, 10857, 3694	0.20700983703136444	2.0080792531371117	154	73	0.474025974025974
8865, 6097, 18750	0.7861345335841179	3.7052246928215027	154	75	0.487012987012987
442, 442, 340	0.93028713110B3436	2.4271275075152516	206	92	0.446601941747572B4
-442, -442, 340	0.46681976318359375	0.9277728423476219	44	21	0.4772727272727273
442, 442, -340	0.661258153617382	1.2051827907562256	44	19	0.4318181818181818
340, 340, 340	0.7760773748159409	1.7507836148142815	108	46	0.42592592592592593
-340, -340, -340	1.0467825457453728	1.1607636734843254	44	17	0.38636363636363635

Σχήμα 18: Μετρήσεις από τα προγράμματα Newton και TriType

Για να υπολογιστούν οι παραπάνω χρόνοι δεν ήταν αρκετή η κλήση μεθόδου της JAVA (currentTimeMillis). Σε μηχάνημα με Windows Me οι μετρήσεις που την χρησιμοποιούν δίνουν αποτελέσματα με ακρίβεια μεγέθους των 50 msec (millisecond). Για περισσότερη ακρίβεια χρησιμοποιήθηκε το JNI (Java Native Interface) με το οποίο κλήθηκε κώδικας γραμμένος σε C που χρησιμοποιεί της μεθόδους QueryPerformanceFrequency και QueryPerformanceCounter. Η τελευταία μάλιστα επιστρέφει τους κύκλους ρολογιού της κεντρικής μονάδας επεξεργασίας (CPU) και είναι σαφώς μεγαλύτερης ακρίβειας από την μέθοδο που παρέχει η ίδια η JAVA.

Παρόλα αυτά όμως είναι φανερό ότι οι χρόνοι που φαίνεται να εκτελούνται τα πρωτότυπα και τα μεταλλαγμένα προγράμματα, δεν είναι εύκολο να μας οδηγήσουν σε κάποια γενικότερη σχέση μεταξύ τους. Σε ορισμένες περιπτώσεις φαίνεται το πρωτότυπο πρόγραμμα, που είναι σαφώς απλούστερης δομής, να απαιτεί χρόνο μεγαλύτερο από το υπέρ-μεταλλαγμένο, κάτι το οποίο μπορεί να οφείλεται στα Thread ή και στην ακρίβεια μέτρησης του χρόνου. Στο σχήμα 19 φαίνονται οι χρόνοι που βρέθηκαν για δέκα εκτελέσεις του πρωτότυπου Newton και του μεταλλαγμένου με τα ίδια δεδομένα. Από το προηγούμενο σχήμα φαίνεται ότι για τα συγκεκριμένα δεδομένα δημιουργούνται 104 μεταλλάξεις από τις οποίες σκοτώνονται οι 79.

Newton_mut

Δεδομένα ελέγχου	Χρόνος Πρωτότυπου	**Χρόνος Μεταλλαγμένου**
34917.18641192952	3.5392815843224525	39.83640355989337
34917.18641192952	1.2697162199765444	2.0793174542486668
34917.18641192952	4.815702576190233	2.1136794108897448
34917.18641192952	5.222179386764765	2.802594745531678
34917.18641192952	4.789721583947539	1.9577934592962265
34917.18641192952	4.86431217379868	10.207177458330989
34917.18641192952	1.1859065685421228	4.795588258653879
34917.18641192952	1.0979064349085093	5.138369735330343
34917.18641192952	4.291054157540202	1.9242695979773998
34917.18641192952	4.895321745425463	2.0407650154083967

Σχήμα 19: Έλεγχος ακρίβειας χρόνων προγράμματος Newton

Έτσι θα πρέπει να αρκεστούμε σε ποιοτική εκτίμηση της μεθόδου. Στον παρακάτω πίνακα φαίνεται το πλήθος των μεταλλάξεων που δημιουργούνται από το Mothra και το εύρος των μεταλλάξεων που δημιουργείται από την παρούσα προσέγγιση για τα αντίστοιχα προγράμματα. Σημειώνεται ότι η σημαντική μείωση του πλήθους των μεταλλάξεων οφείλεται στην πολιτική επιλογής τελεστών μετάλλαξης, όπως έχει περιγραφεί νωρίτερα, στα ιδιαίτερα χαρακτηριστικά της γλώσσας και στην εκμετάλλευση της ροής εκτέλεσης του προγράμματος. Σύμφωνα με το τελευταίο, μια μετάλλαξη δεν δημιουργείται παρά μόνο όταν φτάσει η ροή του προγράμματος στο σημείο που πρόκειται αυτή να εφαρμοστεί.

Πρόγραμμα	Ισχυρή	Προτεινόμενη
Newton [UOH93]	385	44 έως 104
TriType [DGK+88, ORZ93]	951	44 έως 206
TriType2 (βλ. Παρ. Α)	?	44 έως 168

Πίνακας 14: Πλήθος δυναμικά δημιουργημένων μεταλλάξεων

Τα παραπάνω έδειξαν το πλήθος των μεταλλάξεων που δημιουργήθηκαν κατά την εκτέλεση ορισμένων προγραμμάτων με δεδομένα. Εφαρμόζοντας τον τύπο που αναπτύξαμε σε προηγούμενη ενότητα της παρούσας εργασίας θα συγκρίνουμε το αναμενόμενο μέγιστο πλήθος των μεταλλάξεων σε σχέση με το πλήθος των μεταλλάξεων που βρέθηκαν με το σύστημα Mothra και με το τετράγωνο των αναφορών σε μεταβλητές (σύμφωνα με τον Acree). Επίσης δίνονται και οι δημιουργημένες μεταλλάξεις από το πείραμα των Offutt, Rothermel, και Zapf [ORZ93]. Τα αποτελέσματα φαίνονται στον πίνακα που ακολουθεί, ενώ τα προγράμματα των οποίων υπολογίστηκε ο μέγιστος αναμενόμενος αριθμός μεταλλάξεων παρατίθενται στο παράρτημα Α. Σημειώνεται ότι δεν ήταν δυνατό να βρεθούν ακριβώς οι ίδιοι αλγόριθμοι όλων των προγραμμάτων που μετρήθηκαν από το Mothra. Μεγάλες αποκλίσεις είναι πιθανό να οφείλονται σε διαφορετική υλοποίηση των προγραμμάτων, ανάλογα με τις ιδιαιτερότητες της γλώσσας, αλλά και σε αλγοριθμικές διαφορές. Υπενθυμίζεται ο τύπος εκτίμησης του πλήθους των μεταλλάξεων:

$$\sum_{t=1}^{T}(VR_t\cdot(V_t-1))+\sum_{l=0}^{L}\left\lceil\sum_{t=1}^{T}(IV_t\cdot V_t)+3\cdot IC\right\rceil_l+8\cdot L+\sum_{s=0}^{S}(2\cdot C_s+1)+$$
$$3\cdot CR+4\cdot AOR+2\cdot LOR+2\cdot BOR+5\cdot ROR$$

Πρόγραμμα	Mothra (22 Τελεστές)	20 Τελεστές	18 Τελεστές	16 Τελεστές	Acree (VarRef[2])	Νέο
BSearch	307	?	?	?	144	84
Bubble	338	277	224	197	256	72
Calendar	3010	2472	1804	738	625	145
Euclid	196	142	119	119	36	36
Insert	460	352	276	206	441	151
Newton	385	?	?	?	196	118
Quad	359	1279	212	200	169	84
TriType	951	810	579	533	1225	253

Πίνακας 15: Εκτίμηση αναμενόμενων μεταλλάξεων με τον τύπο για το νέο σύστημα

Στον παραπάνω πίνακα η εκτίμηση του Acree επειδή είχε οριστεί για πιο απλές γλώσσες, όπως οι Fortran και η Ada, όπου εφαρμόστηκε αρχικά η μέθοδος των μεταλλάξεων ίσως να έδινε μεγαλύτερα νούμερα αν τα προγράμματα είχαν υλοποιηθεί στις αντίστοιχες γλώσσες. Παρόλα αυτά είναι φανερό συγκριτικά πως η μέθοδος που προτείνεται από την παρούσα εργασία μειώνει σημαντικά το πλήθος των δημιουργούμενων μεταλλάξεων. Επιπλέον σημειώνεται ότι δεν είχε νόημα να μετρηθούν οι βαθμοί μετάλλαξης για κάθε πρόγραμμα γιατί πέρα από πιθανές διαφορές στα ίδια τα προγράμματα η λογική του Mothra (από όπου και οι μετρήσεις) είναι διαφορετική. Στο εν λόγω σύστημα το πλήθος των «σκοτωμένων μεταλλάξεων» υπολογίζεται αθροιστικά, δηλαδή σε κάθε δοκιμή όσες μεταλλάξεις εξοντωθούν προστίθενται στις ήδη «σκοτωμένες». Στην προτεινόμενη προσέγγιση αντιθέτως για καλύτερη αξιολόγηση των δεδομένων εξετάζεται πόσες μεταλλάξεις μπορεί να εξοντώσει κάθε σενάριο από μόνο του.

7.4. Επεκτάσεις τεχνικής – Μελλοντική μελέτη

Αυτό που θα είχε ενδιαφέρον είναι να εξεταστεί το κατά πόσο ο συνδυασμός **αναλυτικής προσέγγισης και εκτέλεσης** κώδικα θα μπορούσε να μειώσει τον απαιτούμενο χρόνο ελέγχου ενός προγράμματος. Θα μπορούσε να γίνει αρχικά αναλυτικός έλεγχος για την αποφυγή δημιουργίας μεταλλάξεων για κάποια

μονοπάτια. Έπειτα ο υπόλοιπος κώδικας να μετασχηματιστεί σύμφωνα με την προτεινόμενη προσέγγιση και να εκτελεστεί (execution-based testing με χρήση σχημάτων). Έτσι θα επιταχυνόταν η διαδικασία μετασχηματισμού του προγράμματος, αφού θα εργαζόταν σε μέρος του και όχι σε όλο. Σε αυτή την προσπάθεια ίσως να ήταν χρήσιμα και τα αποτελέσματα από την κάλυψη μονοπατιών. Θα μπορούσε το σύστημα μετασχηματισμού να γνωρίζει πότε κάποιος κύκλος βρίσκεται σε αδιάβατο μονοπάτι, ώστε να αποφύγει την δημιουργία μεταλλάξεων και του αντίστοιχου σχήματος για αυτό το κομμάτι του προγράμματος. Αυτό δεν θα είχε σημαντικό όφελος για το χρόνο εκτέλεσης του υπερ-μεταλλαγμένου προγράμματος, αλλά θα βελτίωνε το χρόνο μετασχηματισμού και μεταγλώττισης. Επίσης θα μπορούσε ίσως να βοηθήσει και στην απάντηση του ερωτήματος σχετικά με την ισοδυναμία εκδόσεων.

Θα ήταν ενδιαφέρον να μελετηθεί η δυνατότητα προσαρμογής στα σχήματα κάποιου είδους ελέγχου βάση περιορισμών (CBT – Constraint-Based Testing) [OP97]. Αυτό που φαίνεται ότι επιδιώκει αυτού του τύπου ο έλεγχος είναι να βρει τα αναγκαία δεδομένα για την εξόντωση της τρέχουσας μετάλλαξης. Κάτι τέτοιο ίσως να βοηθούσε στην αυτοματοποιημένη αξιολόγηση πιθανών ισοδύναμων μεταλλάξεων αν εφαρμοζόταν μέσα στο κάθε σχήμα. Αν εφαρμοζόταν αυτό το είδος ελέγχου, τότε η αξιολόγηση των αποτελεσμάτων από την χρήση του θα είχε ενδιαφέρον. Θα προέκυπταν συμπεράσματα ακόμα και για την αξία και την αποτελεσματικότητα της παρούσας προσέγγισης σε σχέση με άλλες προσεγγίσεις που φαίνεται σύμφωνα με τους Offutt και Pan ότι έχουν ανάγκη την χρήση CBT.

Ίσως η χρήση στατιστικής και μηχανικής μάθησης για την δημιουργία ομάδων ("clusters") ισοδύναμων μεταλλάξεων να βοηθήσει στην επιτάχυνση της διαδικασίας του ελέγχου. Αυτό σημαίνει και δυναμική δημιουργία πολλών σχημάτων που στην παρούσα προσέγγιση έχουν δημιουργηθεί στατικά, όπως για την μετάλλαξη αριθμητικών, σχεσιακών και λογικών τελεστών. Κάτι τέτοιο δεν είναι δύσκολο, αφού η δομή και οι έλεγχοι που υπάρχουν σε αυτά μπορούν να μοντελοποιηθούν. Τα αποτελέσματα από την εκπαίδευση του συστήματος θα οδηγούσαν το χρήστη του στην εφαρμογή τελεστών μετάλλαξης που έχουν «αξία» για το συγκεκριμένο υπό έλεγχο πρόγραμμα. Αυτό θα σήμαινε μια επιλεκτική μετάλλαξη προσαρμοσμένη όμως στο συγκεκριμένο πρόγραμμα και στα δεδομένα που το ελέγχουν. Πιθανότατα σταδιακά αυτή η αξιολόγηση να οδηγήσει στην εφαρμογή των ίδιων τελεστών με

αυτούς που προτείνονται από τα συστήματα για Ada και Fortran. Μάλιστα στην βιβλιογραφία φάνηκαν προσπάθειες για εφαρμογή στρατηγικών αξιολόγησης των τελεστών μετάλλαξης είτε στον έλεγχο μονάδων [ORZ93, VM98, VMBD01] είτε και στον έλεγχο διεπαφών στη φάση της ολοκλήρωσης συστημάτων [VMBD01]. Η διαφορά θα έγκειται στο ότι το δείγμα εκπαίδευσης του συστήματος θα είναι δυναμικό και κατά συνέπεια και η επιλογή θα μπορεί να αλλάξει οποιαδήποτε στιγμή.

Συνδυασμός program slicing [OC94, Wei84, HHD99] ίσως να μπορούσε να επιφέρει καλύτερες ταχύτητες δημιουργίας δεδομένων ελέγχου και ταυτόχρονα γρηγορότερη εξόντωση μεταλλαγμένων εκδόσεων του υπό έλεγχο προγράμματος. Για παράδειγμα καθώς ο parser εξετάζει και μετασχηματίζει το πρόγραμμα, θα δημιουργεί και καινούρια μικρότερα προγράμματα επικεντρωμένα σε όσο το δυνατό λιγότερες μεταβλητές. Αυτό μπορεί να σημαίνει διπλό πέρασμα για να βρεθούν πρώτα οι μεταβλητές του προγράμματος και έπειτα να σχηματιστούν τα επιμέρους προγράμματα (slices) και κατά συνέπεια αργότερο μετασχηματισμό και μετάφραση του προγράμματος. Αυτό ίσως σε επίπεδο κλάσεων να σημαίνει ότι κάθε κατάσταση θα αποτελείται από λιγότερες μεταβλητές, άρα και όποιοι έλεγχοι μετά από εκτέλεση κάποιας μετάλλαξης είναι λιγότεροι. Επίσης, η αντικατάσταση αναφοράς μεταβλητής θα οδηγεί σε λιγότερες πιθανές περιπτώσεις, αφού θα περιέχονται σε κάθε κομμάτι λιγότερες μεταβλητές. Αυτό θα σημαίνει και έλεγχο περιπτώσεων με λάθος αναφορά μεταβλητής που σχετίζεται πιο άμεσα με το κομμάτι υπό έλεγχο.

Επίσης είναι καλό να εξεταστεί (αφού δημιουργηθεί το σύστημα) το κατά πόσο είναι χρήσιμο να υποστηριχτούν χαρακτηριστικά στην γλώσσα (JAVA), τα οποία όμως δεν αναγνωρίζονται από τον επίσημο μεταφραστή της. Για παράδειγμα για την εφαρμογή της firm μετάλλαξης θα βοηθούσε η χρήση κλήσεων goto και label για την αλλαγή της ροής του προγράμματος με απλή σχετικά σύνταξη. Βέβαια το ίδιο αποτέλεσμα μπορεί να επιτευχθεί και με τις υπάρχουσες επαναληπτικές δηλώσεις, απλά θα πρέπει να χρησιμοποιεί περίπλοκες συνθήκες και πολλά φωλιασμένα loop. Η χρήση αυτών των χαρακτηριστικών δεν είναι απαραίτητο να υποστηρίζεται από το πραγματικό μεταφραστή της γλώσσας. Αρκεί να έχει υλοποιηθεί από το μεταφραστή του συστήματος μετάλλαξης. Θα ήταν ωφέλιμο ίσως να μελετηθεί η χρησιμότητα προσθήκης χαρακτηριστικών που αξιοποιούνται από διάφορες τεχνικές ελέγχου που εφαρμόζουν την μετάλλαξη και δεν υπάρχουν στην έκδοση της JAVA. Για παράδειγμα σε πολλές εργασίες φαίνεται ότι οι εντολές assert σε όσες γλώσσες

επιτρέπεται οδηγεί στην δημιουργία αποτελεσματικότερων δεδομένων ελέγχου, αλλά και στην αξιολόγηση των πιθανών ισοδύναμων μεταλλαγμένων εκδόσεων.

Μεγάλο θα ήταν και το ενδιαφέρον να μελετηθεί ο τρόπος που θα μπορούσε να εφαρμοστεί ένα σχήμα με εμφυτευμένο εσωτερικά έλεγχο, στην δοκιμή διεπαφών στην φάση της ολοκλήρωσης συστημάτων, όπως και στον έλεγχο ιδιαίτερων χαρακτηριστικών των αντικειμενοστρεφών γλωσσών. Οι δύο αυτές περιοχές έχουν αρκετό ενδιαφέρον γιατί σχετίζονται με νέες ιδέες στο χώρο της τεχνολογίας λογισμικού. Η πρώτη περίπτωση είναι σημαντική καθώς διαφαίνεται να καθιερώνεται η τακτική επαναχρησιμοποίησης έτοιμων συστημάτων, υποδομής (legacy systems) ή μη, για μείωση του κόστους και υλοποίηση του συνολικού συστήματος σε λιγότερο χρόνο. Όσο για την μετάλλαξη αντικειμενοστρεφών χαρακτηριστικών αν και περιέχει κάποια στοιχεία από την μετάλλαξη διεπαφών διαφέρει σε ένα κρίσιμο σημείο. Ο έλεγχος ενός προγράμματος γραμμένου από την αρχή σε μια τέτοια γλώσσα μπορεί να ελέγξει καλύτερα τις διεπαφές, γιατί τις αναγνωρίζει στην διάρκεια της μετάφρασης του. Αυτό είναι αδύνατο να γίνει αυτοματοποιημένα για οποιαδήποτε δύο υποσυστήματα που έχουν υλοποιηθεί από διαφορετικούς κατασκευαστές σε διαφορετικές ή μη γλώσσες. Συνήθως αυτά τα συστήματα είναι μαύρα κουτιά για τον ελεγκτή. Μάλιστα θα μπορούσε να ειπωθεί ότι η μετάλλαξη διεπαφών είναι τεχνική «μαύρου κουτιού» που στην περίπτωση εφαρμογής της σε μια οντοκεντρική γλώσσα μπορεί να προσαρμοστεί δυναμικά με βάση τις λεπτομέρειες για την εσωτερική δομή κάθε κλάσης, που αντιστοιχεί με ένα υποσύστημα στην περίπτωση της ολοκλήρωσης.

Τέλος το κομμάτι του μετασχηματισμού των loop σε κλήσεις συναρτήσεων ίσως να μπορεί να βοηθήσει σαν τεχνική γενικότερα, ώστε να είναι πιο εύκολη και αποτελεσματική η συντήρηση του προγράμματος. Μάλιστα, ίσως να βοηθούσε από μόνο του ως εργαλείο συντήρησης, μετασχηματίζοντας τον κώδικα που καταλήγει σε προϊόν στην αγορά. Βέβαια κάτι τέτοιο θα έπρεπε να αντιμετωπίσει το γεγονός ότι μέσα σε μια επαναληπτική δήλωση αλλάζει μέρος της μνήμης του προγράμματος και όχι μόνο μια μεταβλητή του. Αυτό δεν είναι πρόβλημα σε γλώσσες που επιτρέπουν την κλήση μεθόδων με πέρασμα παραμέτρων των αναφορών μεταβλητών (by reference) και όχι των τιμών τους (by value). Για παράδειγμα, θα μπορούσε να γίνει για προγράμματα σε C όχι όμως και στην περίπτωση της JAVA που δεν επιτρέπει αναφορικό πέρασμα μεταβλητών.

Επίσης ένα γρήγορο σύστημα εφαρμογής μετάλλαξης θα είχε αξία για εκπαιδευτικούς σκοπούς. Θα μπορούσε να χρησιμοποιείται από μαθητές και φοιτητές, που αναζητούν την σωστή έκδοση γνωρίζοντας το αποτέλεσμα του προγράμματος που έγραψαν. Αυτό θα βοηθούσε την μελέτη τους χωρίς την παρακολούθηση δασκάλου.

Αναφορές

Βιβλία

[Gos96] Gosling J. et al., "The Java Language Specification", *Addison-Wesley*, 1996.

[LC99] Lemay L. and Cadenhead R., "SAMS Teach Yourself Java 2 in 21 days", *Professional Reference Edition, SAMS Publishing*, 1999.

[VM98] Voas J. M. and McGraw G., "Software fault injection ", *Wiley*, USA 1998

[VMSpec] T. Lindholm, F. Yellin."The JavaTM Virtual Machine Specification". Second Edition, Addison-Wesley, 1996

Ερευνητικές εργασίες (Reports, MSc/PhD thesis)

[Acr80] A.T. Acree. "On Mutation". *PhD thesis*, Georgia Institute of Technology, Atlanta GA, 1980.

[ABD+79] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton and F.G. Sayward. "Mutation analysis". *Technical report* GIT-ICS-79/08, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, September 1979.

[ADH+89] H. Agrawal, R.A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur and E.H. Spafford, "Design of Mutant Operators for the C Programming Language," *Technical Report, SERC-TR-41-P* 1989, Software Engineering Research Center, Purdue University.

[BS79] D. Baldwin and F. Sayward. "Heuristics for determining equivalence of program mutations" *research report* 276, Department of Computer Science, Yale University, 1979.

[HM90] J.R. Horgan and A.P. Mathur. "Weak mutation is probably strong mutation". Technical report SERC-TR-83-P, Software Engineering Research Center, Purdue University, West Lafayette IN, December 1990.

[Kra91] E.W. Krauser. "Compiler Integrated Software Testing". *PhD Thesis (in Partial Fulfillment of the Requirements of the Degree of Doctor of Philosophy)*, Purdue University, December 1991.

[OVP96] A. Jefferson Offutt, Jeff Voas, Jeff Payne. "Mutation Operators for Ada". *Technical Report ISSE-TR-96-09*, Information and Software Systems Engineering, George Mason University. October 1996.

[Tan81] A. Tanaka. "Equivalence testing for FORTRAN mutation system using data flow analysis". *Master's thesis*, School of Information and Computer Science, Georgia Institute of Technology, Atlanta GA, 1981.

[Won93] W.E. Wong. "On Mutation And Data Flow". *PhD Thesis (in Partial Fulfillment of the Requirements of the Degree of Doctor of Philosophy)*, Purdue University, December 1993

[Woo88] M.R. Woodward. "Aspects of path testing and mutation testing". Report 88/2 – November 1988

[Woo89] M.R. Woodward. "Mutation testing of Algebraic Specifications". Report 89/2 – April 1989

Δημοσιεύσεις (Papers)

[BA82] T.A. Budd and D. Angluin. "Two notions of correctness and their relation to testing". *Acta Informatica*, vol. 18: 31-45, November 1982.

[BK88] O. Baruch and S. Katz. "Partially Interpreted Schemas for CSP Programming". Science of Computer Programming, 10(1), pp. 1-18, February 1988.

[DF94] R.K. Doong and P.G. Frankl. "The ASTOOT Approach to Testing Object-Oriented Programs". *ACM Transactions on Software Engineering and Methodology*, vol. 3, no.2, pp. 101-130, April 1994.

[DLS78] R.A. DeMillo, R.J. Lipton, and F.G. Sayward. "Hints on test data selection: Help for the practicing programmer". *IEEE Computer*, vol. 11(4): 34-41, April 1978.

[DO91] R.A. DeMillo, and A.J. Offutt. "Constraint-based automatic test data generation". *IEEE Transactions on Software Engineering,* vol. 17(9): 900-910, September 1991.

[DMM01] M.E. Delamaro, J.C. Maldonado and A.P. Mathur. "Interface Mutation: An Approach for Integration Testing". *IEEE Transactions on Software Engineering,* vol. 27(3), March 2001.

[GM01] S. Ghosh and A.P. Mathur. "Interface Mutation". *Software Testing, Verification and Reliability,* 11: 227-247, 2001

[GOH92] R.Geist, A. J.Offutt, and F.Harris. "Estimation and enhancement of real-time software reliability through mutation analysis". *IEEE Transactions on Computers,* vol. 41(5): 550-558, May 1992. Special issue on Fault-Tolerant Computing.

[Ham77] R.G. Hamlet. "Testing programs with the aid of a compiler". *IEEE Transactions on Software Engineering*, 3(4), July 1977.

[HHD99] R. Hierons, M. Harman, and S. Danicic. "Using program slicing to assist in the detection of equivalent mutants". *Software Testing, Verification and Reliability,* vol. 9(12): 233-262, December 1999

[HOT94] M.J. Harrold, A.J. Offutt and K. Tewary. "An approach to Fault Modeling and Fault Seeding Using the Program Dependence Graph". A preliminary presentation of this work appeared as "Fault Modeling using the Program Dependence Graph" by K. Tewary and M. J. Harrold in *Proceedings of International Symposium on Software Reliability '94 (ISSRE'94), November 1994.*

[How82] W.E. Howden. "Weak mutation testing and completeness of test sets". *IEEE Transactions on Software Engineering,* vol. 8(4): 371-379, July 1982.

[JES98] B.F. Jones, D.E. Eyes and H.H Sthamer. "A Strategy for Using Genetic Algorithms to Automate Branch and Fault-Based Testing". *The Computer Journal*, 41(2), pp. 98-107, 1998.

[KO91] K.N. King and A.J. Offutt. "A Fortran language system for mutation-based software testing". *Software-Practice and Experience,* 21(7), pp. 685-718, July 1991.

[KT94] S. Kirani and W.T. Tsai. "Method Sequence Specification and Verification of Classes". *Journal of Object Oriented Programming,* pp. 28-38, October 1994.

[Mar93] B. Marick. "Using Weak Mutation with GCT". *Testing Foundations*, Champaign Illinois, 1993.

[Mar89] R.J. Martin. "Using the MOTHRA Software Testing Environment to Ensure Software Quality". 1989 IEEE

[MB99] E.S. Mresa and L. Bottaci. "Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study", *The Journal of Software Testing, Verification and Reliability*, 9(4), pp. 205-232, 1999.

[Mor90] L.J. Morell. "A theory of fault-based testing". *IEEE Transactions on Software Engineering*, 16(8), pp. 844-857, August 1990.

[OC94] A.J. Offutt and W.M. Craft. "Using compiler optimization techniques to detect equivalent mutants". *The Journal of Software Testing, Verification and Reliability,* vol. 4(3): 131-154, September 1994.

[Off92] A.J. Offutt. "Investigations of the software testing coupling effect". *ACM Transactions on Software Engineering Methodology,* vol. 1(1): 3-18, January 1992.

[OP97] A.J. Offutt and J. Pan. "Automatically Detecting Equivalent Mutants and Infeasible Paths". *The Journal of Software Testing, Verification and Reliability,* vol. 7(3): 165-192, September 1997.

[OL94] A.J. Offutt and A. Lee. "An Empirical Evaluation of Weak Mutation". *IEEE Transactions on Software Engineering, 20(5) 337-344,* May 1994.

[OLR+96] 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,* vol. 5(2): 99-118, April 1996.

[PHP99] R.P. Pargas, M.J. Harrold and R.R. Peck. "Test-Data Generation Using Genetic Algorithms". *The Journal of Software Testing, Verification and Reliability,* 1999.

[SH99] S. Sinha and M.J. Harrold. "Criteria for Testing Exception-Handling Constructs in Java Programs". *Technical Report OSU-CISRC-6/99-TR16, June 1999.*

[Spa90] E. Spafford. "Extending Mutation Testing to Find Environmental Bugs". *Software Practice & Experience*, 20(2), pp. 181-189, February 1990.

[VMBD01] A.M.R. Vincenzi, J.C. Maldonado, E.F. Barbosa and M.E. Delamaro. "Unit and integration testing strategies for C programs using mutation". *The Journal of Software Testing, Verification, and Reliability,* vol. 11:249-268, 2001

[Wah95] K.S.H.T. Wah. "Fault coupling in finite bijective functions". *The Journal of Software Testing, Verification, and Reliability,* vol. 5(3): 3-47, March 1995.

[Wah00] K.S.H.T. Wah. "A theoretical study of fault coupling". *The Journal of Software Testing, Verification, and Reliability,* vol. 10(3): 3-46, March 2000.

[Wei84] M. Weiser. "Program Slicing". *IEEE Transactions on Software Engineering*, 10(4): 352-357, July 1984.

[WF93] S.N. Weiss, V.N. Fleyshgakker, "Improved serial algorithms for mutation analysis", *ACM SIGSOFT Software Engineering Notes*, v.18 n.3, p.149-158, July 1993

Πρακτικά (Proceedings)

[BGA01] J.M. Bieman, S. Ghost and R.T. Alexander. "A Technique for Mutation of Java Objects". In *Proceedings Automated Software Engineering 2001 Conference (ASE 2001)*, November 2001

[BHJT] B. Baudry, Y. Le Traon, J.M. Jézéquel, and H. Vu Le. "Trustable components: Yet another mutation-based approach". In *Proceedings of the 1st Symposium on Mutation Testing (Mutation'2000)*, pages 69--76, San Jose, CA, 2000.

[BOP00] U. Buy, A. Orso and M. Pezze. "Automated Testing of Classes". In *Proceedings of the International Symposium on Software Testing and Analysis 2000*, Portland, Oregon, United States.

[BOY00] P.E. Black, V. Okun and Y. Yesha. "Mutation Operators for Specifications". In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering, 2000 (ASE 2000)*, pp. 81–88.

[Dem89] R.A. DeMillo. "Test Adequacy and Program Mutation". In *Proceedings of the 11th international conference on Software engineering,* May 1989

[DGK+88] R.A. DeMillo, D.S. Guindi, K.N. King, W.M. McCracken, and A.J. Offutt. "An extended overview of the Mothra software testing environment". In *Proceedings of the Second Workshop on Software Testing, Verification and Analysis,* pp. 142-151, Banff Alberta, July 1988, IEEE Computer Society Press.

[DKM91] R. A. DeMillo, E. W. Krauser, and A. P. Mathur. Compiler-integrated program mutation, In *Proceedings of the 15th Annual Computer Software and Applications Conference*, Tokyo, Japan, September 1991. Kogakuin University.

[FW94] V.N. Fleyshgakker and S.N. Weiss. "Efficient Mutation Analysis: A New Approach". In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 94), (Seattle, WA),* pp. 185-195, ACM SIGSOFT, ACM Press, August 1994.

[GW85] M.R. Girgis and M.R. Woodward. "An integrated system for program testing using weak mutation and data flow analysis". In Proceedings of the *Eighth International Conference on Software Engineering*, pp. 313-319, London UK, August 1985. IEEE Computer Society.

[Hay94] J.H. Hayes. "Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach". In *Proceedings, Object-Oriented Methodologies and Systems*, E. Bertino, S. Urban (Eds.), LNCS 858. Springer-Verlag, Berlin 1994.

[KCM99] S. Kim, J. Clark, J. McDermid, "Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation", *28 JAIIO: Symposium on Software Technology (SoST'99)*, Sept. 1999.

[KCM00] S. Kim, J. Clark, J. McDermid, "Class Mutation: Mutation Testing for Object-Oriented Programs", In *Proceedings of the ObjectDays 2000, Germany, October 2000.*

[KCM00b] S. Kim, J. Clark, J. McDermid, "Investigating the Effectiveness of Object-Oriented Testing Strategies with the Mutation Method". In *Proceedings of the ObjectDays 2000*. October 2000.

[KSG+94] D. Kung, N. Sushak, J. Gao, P. Hsia, Y. Toyoshima and C. Chen. "On Object State Testing". In *Proceedings of IEEE COMPSAC'94,* pp. 222-227, IEEE Computer Society Press, 1994.

[Mar91] B. Marick. "The weak mutation hypothesis". In Proceedings of the *Third Symposium on Software Testing, Analysis and Verification*, pp. 190-199, Victoria, British Columbia, Canada, October 1991, IEEE Computer Society.

[Off95] A.J. Offutt. "A Practical System for Mutation Testing: Help for the Common Programmer". In *12th International Conference on Testing Computer Software*, pages 99--109, Washington, DC, June 1995.

[ORZ93] A.J. Offutt, G. Rothermel, and C. Zapf. "An experimental evaluation of selective mutation". In *Proceedings of the Fifteenth International Conference on Software Engineering, (Baltimore, MD)*, pp. 100-107, IEEE Computer Society Press, May 1993.

[OU00] A.J. Offutt and R.H. Untch. "Mutation 2000: Uniting the Orthogonal". In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, 45--55, San Jose, CA, October 2000.

[YCJ98] H. Yoon, B. Choi and J.O. Jeon. "Mutation-based Inter-class Testing". In *Proceedings of Asia Pacific Software Engineering Conference (APSEC '98)*, pages 174–181. IEEE Computer Society Press, Los Alamitos, California, 1998.

[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,* pp. 139-148, Cambridge MA, June 1993.

[WDM94] W.E. Wong, M.E. Delamaro, J.C. Maldonado, and A.P. Mathur. "Constrained mutation in C programs". In *Proceedings of the 8th Brazilian Symposium on Software Engineering, (Curitiba, Brazil),* pp. 439-452, October 1994.

[WH88] M.R. Woodward and K. Halewood. "From weak to strong, dead or alive? An analysis of some mutation testing issues". In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis,* pp. 152-158, Banff Alberta, July 1988. IEEE Computer Society Press.

[WHHR88] D. Wu, M.A. Hennell, D. Hedley and I.J. Riddell. "A practical method for software quality control via program mutation". In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis,* pp. 159-170, Banff Alberta, July 1988. IEEE Computer Society Press.

Διαδίκτυο (URL's)

http://www.javaworld.com/

http://java.sun.com/

Παράρτημα Α

Εδώ παρατίθενται τα προγράμματα με τα οποία δοκιμάστηκε η τεχνική της εφαρμογής των σχημάτων μετάλλαξης, όπως αυτή προτάθηκε και περιγράφηκε από την παρούσα εργασία.

Το πρόγραμμα Newton

Το πρόγραμμα αυτό υπολογίζει την τετραγωνική ρίζα κάποιου θετικού αριθμού. Ο κώδικας του σε γλώσσα JAVA είναι αντίστοιχος με αυτό που βρέθηκε στην αρθρογραφία [UOH93]:

```
public double Newton(double num) {
 double res, NewGuess, d, e;
 e = 0.001;
 NewGuess = (num / 2.0) + 1.0;
 res = 0.0;
 d = NewGuess - res;
 while (d > e) {
    res = NewGuess;
    NewGuess = (res + (num / res)) / 2;
    d = NewGuess - res;
    if (d < 0.0) d = - d;
 }
 return res;
}
```

Το πρόγραμμα TriType

Το πρόγραμμα αυτό υπολογίζει το είδος κάποιου τριγώνου έχοντας σαν εισόδους τα μήκη των πλευρών του. Για λόγους απλότητας και μόνο και χωρίς να βλάπτεται η γενικότητα τα μήκη των πλευρών που απαιτούνται είναι ακέραιες τιμές. Ο κώδικας στη JAVA είναι όπως ακολουθεί και έχει ακριβώς την ίδια λογική με αντίστοιχο πρόγραμμα σε Fortran [DGK+88]. Αυτό που χρειάζεται να σημειωθεί

είναι ότι η μεταβλητή tritype ανήκει στην κλάση που περιέχει την μέθοδο, αλλά θεωρείται μεταβλητή του προγράμματος. Το πρόγραμμα TriType2 είναι παρόμοιο με την μόνη διαφορά ότι κάποιοι έλεγχοι άλλαξαν για να χρησιμοποιηθούν δηλώσεις switch-case.

```
public void TriType(int i, int j, int k) {
    if (i < 0 || j < 0 || k < 0) {
       tritype = 4;  return;
    }
    tritype = 0;
    if (i == j) tritype++;
    if (i == k) tritype = tritype + 2;
    if (j == k) tritype += 3;
 /* Confirm it is legal triangle */
    if (tritype == 0) {
      if (  i + j < k ||  j + k < i  || i + k < j)
              tritype = 4;
       else tritype = 1;
       return;
    }
    if (tritype > 3) tritype = 3;
    else if ( tritype == 1 && i + j > k)
              tritype = 2;
    else if ( tritype == 2 && i + k > j)
              tritype = 2;
    else if (tritype == 3 && j + k > i)
              tritype = 2;
    else tritype = 4;
  }
```

```
public void TriType2(int i, int j, int k) {
    if (i < 0 || j < 0 || k < 0) {
       tritype = 4;  return;
    }
    tritype = 0;
    if (i == j) tritype++;
    if (i == k) tritype = tritype + 2;
    if (j == k) tritype += 3;
 /* Confirm it is legal triangle */
    if (tritype == 0) {
       if (  i + j < k ||  j + k < i  || i + k < j)
              tritype = 4;
       else tritype = 1;
       return;
    }
     /* Check the tritype now */
    int tmp_type = 4;
    switch (tritype) {
       case 1:  if ( i + j > k)
                   tmp_type = 2; break;
       case 2:  if ( i + k > j)
                   tmp_type = 2; break;
       case 3: if ( j + k > i)
                   tmp_type = 2; break;
       default: // means tritype > 3
          tmp_type = 3; break;
    } tritype = tmp_type;
  }
```

Το πρόγραμμα BSearch

Το πρόγραμμα αυτό βρίσκει αν υπάρχει στοιχείο με τιμή ίση του a σε ένα ταξινομημένο πίνακα, ξεκινώντας από την θέση low μέχρι την θέση high. Ο κώδικας στη JAVA έχει και εδώ ακριβώς την ίδια λογική με αντίστοιχο πρόγραμμα όπως αυτό αναφέρεται στην αρθρογραφία [DGK+88]:

```
public bool BSearch(int[] arr, int low, int high, int a) {
int mid;
while(high >= low) {
   mid = (low + high)/2;
   if (a == arr[mid]) return true;
   else if (a > arr[mid]) low = mid + 1;
   else high = mid - 1;
}
return false;
}
```

Το πρόγραμμα Bubble

Ο αλγόριθμος Bubble είναι ιδιαίτερα γνωστός και χρησιμοποιείται για την ταξινόμηση στοιχείων ενός πίνακα. Το παρακάτω πρόγραμμα τον υλοποιεί σε JAVA με τρόπο τέτοιο ώστε να μην διαφέρει με υλοποιήσεις σε γλώσσες που ο πίνακας δεν είναι οντότητα. Σε αντίθετη περίπτωση δεν χρειαζόταν η παράμετρος length γιατί αποτελεί μέλος της κλάσης των πινάκων (a.length).

```
void sort(int a[], int length) {
     int i, j, T;
  for (i = 0; i < length; i++)
     for (j = length-1; j > i; j--) {
        if (a[j-1] > a[j]) {
             T = a[j-1];
             a[j-1] = a[j];
             a[j] = T;
        } } }
```

Το πρόγραμμα Euclid

Αυτό είναι ένα πρόγραμμα που εφαρμόζει τον αλγόριθμο του Ευκλείδη για την εύρεση του μέγιστου κοινού διαιρέτη μεταξύ δύο αριθμών.

```
long Eucleid_gcd( long m, long n ) {
 while( n!=0) {
  long rem = m % n;
  m = n;
  n = rem;
 } return m;
}
```

Το πρόγραμμα CALENDAR

Περνώντας τον αριθμό του αιώνα (century), του έτους (year, 0-99), του μήνα (month) και της μέρας (day) στο πρόγραμμα calendar υπολογίζονται οι μέρες από μια απομακρυσμένη ημερομηνία. Αφού αυτό γίνει για κάθε ημερομηνία τότε αφαιρούνται οι μέρες και επιστρέφεται το αποτέλεσμα.

```
long calendar (long[2] century, long[2] year, long[2] month,
long[2] day) {
   long[] dayN = new long[2];
dayN[0] = dayN[1] = 0;
for (int I = 0; I < 2; I++) {
   if (month[I] > 2) month[I] -= 3;
   else {
      month[I] += 9;
      if (year[I]) year[I]--;
      else {
         year[I] = 99; century[I]--;
       }}
 dayN[I] = ((146097L * century[I]) / 4L + (1461L*year[I]) / 4L
+ (153L   * month[I] + 2L) / 5L + day[I]  + 1721119L);
}
  return dayN[1]-dayN[0];}
```

Το πρόγραμμα INSERT

Με αυτό το πρόγραμμα έχοντας ένα ταξινομημένο πίνακα ακεραίων αναζητούμε θέση για να εισάγουμε την τιμή a. Αν αυτό το στοιχείο υπάρχει ήδη στον πίνακα, τότε δεν το εισάγουμε. Επίσης υποθέτουμε ότι ο πίνακας είναι μεγέθους (length+1).

```
public void Insert(int[] arr, int low, int high, int a) {
int mid, length;
length = high;
boolean notFound = true;
while(high >= low && notFound) {
   mid = (low + high)/2;
   if (a == arr[mid]) notFound = false;
   else if (a > arr[mid]) low = mid + 1;
   else high = mid - 1;
}
if (notFound) {
      for (int I = length; I > high + 1; I--)
            arr[I] = arr[I-1];
      arr[high+1] = a;}}
```

Το πρόγραμμα QUAD

Αυτό είναι ένα πρόγραμμα που υπολογίζει τις ρίζες μιας τετραγωνικής εξίσωσης της μορφής $ax^2+bx+c=0$. Στο τέλος επιστρέφει ένα διθέσιο πίνακα με τις λύσεις.

```
double[2] Quad(double a, double b, double c) {
double[] res = new double[2];
res[0] = 0.0; res[1] = 0.0;
double D = b*b – 4*a*c;
if (D >= 0) {
res[0] = (-b + sqrt(D)) / 2*a;
res[1] = (-b - sqrt(D)) / 2*a;
} return res;}
```

Παράρτημα Β

Παρακάτω ακολουθεί μέρος του κώδικα από το πρότυπο, που υλοποιήθηκε στα πλαίσια της παρούσας εργασίας, για να δοκιμαστεί η προτεινόμενη μέθοδος. Συγκεκριμένα δίνεται η υλοποίηση της μεθόδου ArithWeakOp που είναι μέλος της κλάσης JavaMutantSchema. Καλώντας την (μέθοδο) εφαρμόζονται όλοι οι τελεστές που σχετίζονται με μετάλλαξη μιας αριθμητικής πράξης. Αναλυτικότερα μεταλλάσσονται:

- η ίδια η πράξη με αντικατάσταση της από άλλους αριθμητικούς τελεστές,
- οι αναφορές στις μεταβλητές που συμμετέχουν με αντικατάσταση άλλων επιτρεπτού τύπου που περιλαμβάνονται στο πρόγραμμα και έχουν περιληφθεί στην βοηθητική λίστα της κλάσης JMSTrace μέχρι την στιγμή της μετάλλαξης,
- η χρήση σταθερών με διαφοροποίηση τους (εδώ επειδή είναι αριθμοί προστίθεται και αφαιρείται το 1)
- η ίδια η τιμή του κάθε τελεσταίου χρησιμοποιώντας την αντίθετη της

Μέσα στην ArithWeakOp υπάρχουν κλήσεις στην ArithOp που είναι μια switch-case συνάρτηση που ανάλογα με τον κωδικό που λαμβάνει ως παράμετρο εκτελεί την αντίστοιχη αριθμητική πράξη μεταξύ των δύο τελεσταίων a και b. Έπειτα συγκρίνει το αποτέλεσμα με αυτό της ArithOp(a, b, orig_op), όπου orig_op ο κωδικός της πράξης στο πρωτότυπο πρόγραμμα. Αν πάλι κληθεί η ArithWeakOp με τουλάχιστον μιας από τις δύο πρώτες παραμέτρους να είναι αλφαριθμητικό και όχι αριθμός, τότε αναζητούνται ανάλογα με το όνομα της μεταβλητής ο τύπος της και έπειτα, αφού γίνει ότι και παραπάνω, δοκιμάζεται η ίδια πράξη έχοντας κάθε φορά διαφορετική μεταβλητή από αυτή που υπάρχει στον πρωτότυπο κώδικα. Αν πάλι ο τελεσταίος της πράξης είναι σταθερή τιμή δοκιμάζονται και πράξεις τροποποιώντας την προσθέτοντας ή αφαιρώντας μια μονάδα. Τέλος σε κάθε περίπτωση και ανεξάρτητα από τις δύο πρώτες παραμέτρους εκτελούνται και πράξεις με τις αντίθετες τιμές τους.

Για καλύτερη κατανόηση του κώδικα και για πιο άπειρους προγραμματιστές σε αντικειμενοστρεφείς γλώσσες γενικότερα, σημειώνεται ότι στην JAVA υπάρχει η δυνατότητα να δηλωθούν στην ίδια κλάση μέθοδοι με το ίδιο όνομα και διαφορετική σειρά παραμέτρων.

```
public double
ArithWeakOp(double a, double b, int cnst, int orig_op, String prefix,
String suffix) {
double res, res_m; int i; JCheckBox tmp;

res = ari.ArithOp(a, b, orig_op);
//Check in order to prevent double check when in loop...
if ((lines_num - 1) > lineno) return res;
i = 0;
while (i < ari.AO_NUM) {
 tmp = (JCheckBox)frame.ari.get(i);
 if (tmp.isSelected() && i != orig_op) {
   res_m = ari.ArithOp(a, b, i);
   AddMutant(lineno, prefix, ari.oper[i], suffix);
   if (res_m != res) {
     ari.KillMutant(a, b, orig_op, i);
     IncrKilledMutants(lineno);
   } else if (ari.IsEquevalent(a, b, orig_op, i))
            IncrEqualMutants(lineno);
 }
 i++;
}

// Check the parameter from interface for ++ and --
if (MutateConst) {
  if (cnst % 2 ==  1) {
  //check (a+1) orig_op b
  res_m = ari.ArithOp(a + 1, b, orig_op);
  AddMutant(lineno, prefix + "+ 1 ", ari.oper[orig_op], suffix);
  if (res_m != res) {
    ari.KillMutant(a + 1, b, orig_op, orig_op);
    IncrKilledMutants(lineno);
  } else if (ari.IsEquevalent(a + 1, b, orig_op, orig_op))
             IncrEqualMutants(lineno);

  //check (a-1) orig_op b
  res_m = ari.ArithOp(a - 1, b, orig_op);
  AddMutant(lineno, prefix + "- 1 ", ari.oper[orig_op], suffix);
  if (res_m != res) {
    ari.KillMutant(a - 1, b, orig_op, orig_op);
    IncrKilledMutants(lineno);
  } else if (ari.IsEquevalent(a - 1, b, orig_op, orig_op))
           IncrEqualMutants(lineno);
}
if (cnst > 0 && cnst % 2 == 0) {
  //check a orig_op (b+1)
  res_m = ari.ArithOp(a, b + 1, orig_op);
  AddMutant(lineno, prefix, ari.oper[orig_op], " 1 +" + suffix);
  if (res_m != res) {
      ari.KillMutant(a, b + 1, orig_op, orig_op);
      IncrKilledMutants(lineno);
  } else if (ari.IsEquevalent(a, b + 1, orig_op, orig_op))
              IncrEqualMutants(lineno);

  //check a orig_op b - 1
  res_m = ari.ArithOp(a, b - 1, orig_op);
  AddMutant(lineno, prefix, ari.oper[orig_op], " -1 +" + suffix);
  if (res_m != res) {
    ari.KillMutant(a, b - 1, orig_op, orig_op);
    IncrKilledMutants(lineno);
```

```
    } else if (ari.IsEquevalent(a, b - 1, orig_op, orig_op))
                IncrEqualMutants(lineno);
  }

  //check -a orig_op b
  res_m = ari.ArithOp(-a, b, orig_op);
  AddMutant(lineno, prefix + "*(-1)", ari.oper[orig_op], suffix);
  if (res_m != res) {
    ari.KillMutant(-a, b, orig_op, orig_op);
    IncrKilledMutants(lineno);
  } else if (ari.IsEquevalent(-a, b, orig_op, orig_op))
              IncrEqualMutants(lineno);

  if (orig_op != ari.AO_ADD && orig_op != ari.AO_SUB) {
    //check a orig_op -b, if orig_op not + or -
    res_m = ari.ArithOp(a, -b, orig_op);
    AddMutant(lineno, prefix, ari.oper[orig_op], "(-1)*" + suffix);

    if (res_m != res) {
      ari.KillMutant(a, -b, orig_op, orig_op);
      IncrKilledMutants(lineno);
    } else if (ari.IsEquevalent(a, -b, orig_op, orig_op))
                IncrEqualMutants(lineno);
  }
  }
  return res;
  }

  // test weak mutation for operation on two double
  public double
  ArithWeakOp(String strA, double b, int orig_op, String prefix, String
  suffix) {

  int varIndex = trace.GetVarIndex(strA);
  double a = trace.GetDblVarVal(varIndex);

  //Check in order to prevent double check when in loop...
  if ((lines_num - 1) > lineno) return ari.ArithOp(a, b, orig_op);

  double res = ArithWeakOp(a, b, 2, orig_op, prefix, suffix);

  if (MutateAssignments) {
    int varsN = trace.GetVarsNum(JMS_DBL);
    double a1, res_m;
    for (int i = 0; i < varsN; i++) {
      if (i != varIndex) {
        a1 = trace.GetDblVarVal(i);
        res_m = ari.ArithOp(a1, b, orig_op);
        AddMutant(lineno, prefix, ari.oper[orig_op], suffix);
        if (res_m != res) {
          ari.KillMutant(a1, b, orig_op, orig_op);
          IncrKilledMutants(lineno);
        } else if (ari.IsEquevalent(a1, b, orig_op, orig_op))
                    IncrEqualMutants(lineno);

      }
    }
  }
  return res;
  }
```

```
public int
IntArithWeakOp(String strA, String strB, int orig_op, String prefix,
String suffix) {

int varIndexA = trace.GetVarIndex(strA);
int a = trace.GetIntVarVal(varIndexA);
int varIndexB= trace.GetVarIndex(strB);
int b = trace.GetIntVarVal(varIndexB);

// Prevent double checks in loop...
if ((lines_num - 1) > lineno) return (int)ari.ArithOp(a, b, orig_op);

int res = (int)ArithWeakOp((double)a, (double)b, 0, orig_op, prefix,
suffix);

if (MutateAssignments) {
  int varsN = trace.GetVarsNum(JMS_INT);
  int a1, res_m;
  for (int i = 0; i < varsN; i++) {
    if (i != varIndexA && i!=varIndexB) {
      a1 = trace.GetIntVarVal(i);
      res_m = (int)ari.ArithOp(a1, b, orig_op);
      AddMutant(lineno, prefix, ari.oper[orig_op], suffix);
      if (res_m != res) {
        ari.KillMutant(a1, b, orig_op, orig_op);
        IncrKilledMutants(lineno);
      } else if (ari.IsEquevalent(a1, b, orig_op, orig_op))
                    IncrEqualMutants(lineno);
      res_m = (int)ari.ArithOp(a, a1, orig_op);
      AddMutant(lineno, prefix, ari.oper[orig_op], suffix);
      if (res_m != res) {
        ari.KillMutant(a, a1, orig_op, orig_op);
        IncrKilledMutants(lineno);
      } else if (ari.IsEquevalent(a, a1, orig_op, orig_op))
              IncrEqualMutants(lineno);
}}}
return res;
        }
```

Εδώ παρατίθεται επίσης μέρος του κώδικα που προέκυψε από την μετάλλαξη της επαναληπτικής δήλωσης στο πρόγραμμα Newton (βλ. Παρ. Α). Για λόγους αποφυγής φλυαρίας δεν περιγράφονται αναλυτικά όλες οι μεταλλάξεις της επαναληπτικής δήλωσης γιατί αυτό κρίνεται όχι ιδιαίτερης σημασίας για τον αναγνώστη και μάλλον θα τον κουράσει παρά θα τον βοηθήσει σε περαιτέρω κατανόηση της παρούσας μελέτης. Υπενθυμίζεται ότι ο κώδικας που ακολουθεί θα πρέπει να δημιουργείται από το μετασχηματιστή του προτεινόμενου συστήματος κατά την διάρκεια της ανάλυσης του κώδικα του προγράμματος.

```
package JMSTests.JMSLoopSchemas;    // This could be missing

import java.lang.*; // System
import java.io.*;  // println
```

```
/* Added imports from mutation compiler */
import javax.swing.*;    // JCheckBox Object
import JMutOps.*;        // Mutation functions

public class Newton_JMSLoop {
    private JavaMutantSchema jms;
    private int lineno;
    private int loops;
    //Loop Call Parameters --> lcp
    private double num_lcp, res_lcp, d_lcp, NewGuess_lcp, e_lcp;
    //Original Loop Call Values --> olcv
    private double num_olcv, res_olcv, d_olcv, NewGuess_olcv, e_olcv;
    //Mutated Loop Call Values --> from the name of the variable and
"_mlcv"
    private double num, res, d, NewGuess, e;

    public void
    InitLoopCall(double num1, double res1, double NewGuess1, double
e1, double d1) {
        InitLoopParams(num1, res1, NewGuess1, e1, d1);
        /* Initialize the mutated loop call values
         when each mutation case is called */
    }
    private void
    InitLoopParams(double num1, double res1, double NewGuess1, double
e1, double d1) {
        d_lcp = d_olcv = d1;
        num_lcp = num_olcv = num1;
        res_lcp = res_olcv = res1;
        NewGuess_lcp = NewGuess_olcv = NewGuess1;
        e_lcp = e_olcv = e1;
    }
    /*
     * Always use it after the InitLoopParams has been called.
     */
    private void
    InitMutLoopParams() {
        d = d_lcp;
        num = num_lcp;
        res = res_lcp;
        NewGuess = NewGuess_lcp;
        e = e_lcp;
    }

    private boolean IsStateEqual() {
        if (res_olcv != res)
            return false;
        return true;
    }
    private void Check(){
        if (IsStateEqual() == false)
            jms.KillLoopCallMutant();

    }

    private void
    Newton_LoopCall1() {

        d_olcv = NewGuess_olcv - res_olcv;
```

```
        loops = 0;
        while (d_olcv > e_olcv) {
            loops++;
            res_olcv = NewGuess_olcv;
            NewGuess_olcv = (res_olcv + (num_olcv / res_olcv)) / 2;
            d_olcv = NewGuess_olcv - res_olcv;
            if (d_olcv < 0.0)
                d_olcv = - d_olcv;
        }
    }


/*
 * Newton_LoopCall1_m1: moved Statement(1) OUTSIDE loop.
 */
    private void
    Newton_LoopCall1_m1() {
        jms.AddLoopMutant(JMSLoopInfo.ML_2_TO_N);
        InitMutLoopParams();
        d = NewGuess - res;
        res = NewGuess;
        for (int JMS_i = 0; d > e && JMS_i < loops; JMS_i++) {
            NewGuess = (res + (num / res)) / 2;
            d = NewGuess - res;
            if (d < 0.0)
                d = - d;
        }
            Check();
    }

/*
 * Newton_LoopCall1_m11:
 * moved Statement(1) OUTSIDE loop.
 * moved Statement(N) OUTSIDE loop.
 */
    private void
    Newton_LoopCall1_m11() {
        jms.AddLoopMutant(JMSLoopInfo.ML_2_TO_Nmin);
        InitMutLoopParams();
        d = NewGuess - res;

        res = NewGuess;
        for (int JMS_i = 0; d > e && JMS_i < loops; JMS_i++) {
            NewGuess = (res + (num / res)) / 2;
            d = NewGuess - res;
        }
        if (d < 0.0)
            d = - d;
                Check();
    }

/*
 * Newton_LoopCall1_m12:
 * moved Statement(1) OUTSIDE loop.
 * moved Statement(N+1) INSIDE loop.
 */
    private void
    Newton_LoopCall1_m12() {
        jms.AddLoopMutant(JMSLoopInfo.ML_2_TO_Nmax);
        InitMutLoopParams();
```

```
        d = NewGuess - res;
        res = NewGuess;
        for (int JMS_i = 0; d > e && JMS_i < loops; JMS_i++) {
            NewGuess = (res + (num / res)) / 2;
            d = NewGuess - res;
            if (d < 0.0)
                d = - d;
        }
                Check();
    }
    private void
    Newton_LoopCall1_m2() {
        jms.AddLoopMutant(JMSLoopInfo.ML_1_TO_Nmin);
        InitMutLoopParams();
        d = NewGuess - res;
        for (int JMS_i = 0; d > e && JMS_i < loops; JMS_i++) {
            res = NewGuess;
            NewGuess = (res + (num / res)) / 2;
            d = NewGuess - res;
        }
        if (d < 0.0)
            d = - d;
                Check();
    }
    private void
    Newton_LoopCall1_m4_1() {
        jms.AddLoopMutant();InitMutLoopParams();

        //d = NewGuess - res;
        d = NewGuess + res;
        for (int JMS_i = 0; d > e && JMS_i < loops; JMS_i++){
            res = NewGuess;
            NewGuess = (res + (num / res)) / 2;
            d = NewGuess - res;
            if (d < 0.0)
                d = - d;
        }
        Check();
    }
```

Και αφού λοιπόν τελειώσει ο ορισμός όλων των πιθανών μεταλλάξεων της επαναληπτικής δήλωσης ορίζεται από τον μετασχηματιστή η παρακάτω διαδικασία που αποτελεί ένα σχήμα που καλεί και συγκρίνει αποτελέσματα όλων των "Newton_LoopCall1_mX" με την πρωτότυπη "Newton_LoopCall".

```
     public double
    Newton_LoopCall1_DoMutation(double num1, double res1, double
NewGuess1, double e1, int startLine, int endLine, JavaMutantSchema
javams) {


        double LoopCallRes;
        JCheckBox tmp;
        jms = javams;
        lineno = startLine;
```

```
            InitLoopCall(num1, res1, NewGuess1, e1, 0.0);
            Newton_LoopCall1();
            jms.AddLoopTrace(startLine + 1, endLine - 1);

           /* ML_2_TO_N- = 5; // created ==> LoopCall1_m11(...) */
            tmp = (JCheckBox) jms.frame.loop.get(2);
            if (tmp.isSelected()) Newton_LoopCall1_m11();

           /* ML_2_TO_N+ = 6; // created ==> LoopCall1_m12(...) */
            tmp = (JCheckBox) jms.frame.loop.get(2);
            if (tmp.isSelected()) Newton_LoopCall1_m12();

            tmp = (JCheckBox) jms.frame.loop.get(2);
            if (tmp.isSelected()) Newton_LoopCall1_m2();

            tmp = (JCheckBox) jms.frame.loop.get(4);
            if (tmp.isSelected()) {

Newton_LoopCall1_m4_1(); Newton_LoopCall1_m4_2();
Newton_LoopCall1_m4_3(); … … … Newton_LoopCall1_m4_81();
Newton_LoopCall1_m4_82(); Newton_LoopCall1_m4_83();
            }
            return res_olcv;
        }
}
```