Θεωρητικές και Εφαρμοσμένες Τεχνικές Ελέγχου σε Σύγχρονα Αντικειμενοστραφή Συστήματα

Οικονομικό Πανεπιστήμιο Αθηνών

Τσίρμπας Δημήτριος, Μανδελιάς Αλέξιος

September 15, 2022

Contents

1 Περίληψη

Η επαλήθευση και η επικύρωση του λογισμικού ήταν ανέκαθεν μια από τις πιο σημαντικές πτυχές της ανάπτυξης λογισμικού. Αυτήν τη στιγμή εκτιμάται ότι πάνω από το 50% του χρόνου ανάπτυξης του λογισμικού καταναλώνεται στη φάση της δοκιμής/επαλήθευσης (testing). Από την άλλη, η επικράτηση των αντικειμενοσταφών γλωσσών προγραμματισμού έχει αλλάξει ριζικά, πέρα από τον τρόπο της σχεδίασης και ανάπτυξης του λογισμικού, και τις απαιτήσεις και δυσκολίες στην διαδικασία της επαλήθευσης, με αποτέλεσμα πολλές παραδοσιακές τεχνικές επαλήθευσης να μην εξυπηρετούν πια τους προγραμματιστές. Για αυτόν τον λόγο έχει διεξαχθεί εντατική έρευνα τα τελευταία χρόνια με σκοπό τόσο να προσδιοριστούν ποιές τεχνικές είναι ακόμα αποτελεσματικές, όσο και να βρεθούν καινούριες, κατάλληλες για τις προκλήσεις των αντικειμενοστραφών συστημάτων. Σε αυτήν την αναφορά ερευνούμε τις δυσκολίες που επιφέρει η χρήση του αντικειμενοστρεφούς μοντέλου στη διαδικασία της επαλήθευσης, τις διαφορές της αντικειμενοστρεφούς επαλήθευσης από την παραδοσιακή (διαδικαστική) και τρόπους συμφιλίωσης των διαφορών αυτών. Αναλύουμε ποιές τεχνικές τελικά άντεξαν την πάροδο του χρόνου και παραμένουν σε χρήση ακόμα, καθώς και καινούριες τεχνικές και εργαλεία που αναπτύχθηκαν για να αντιμετωπίσουν τις σύγχρονες προκλήσεις.

2 Διαδικαστικό vs Αντικειμενοστραφές μοντέλο επαλήθευσης

Οι κύριες διαφορές μεταξύ των μεθόδων επαλήθευσης των δύο μοντέλων ανάπτυξης συνοψίζονται στην εικόνα 2 (Hayes and Huffman [8] όπως αναφέρεται από τους Gordon and Roggio [7]).

Ο πίνακας αυτός αναφέρεται σε 5 από τα 11 αξιώματα του Haeys, καθώς τα υπόλοιπα είναι κοινά μεταξύ των μοντέλων ανάπτυξης, σύμφωνα με τον συγγραφέα. Μπορούμε να εκτιμήσουμε λοιπόν ότι οι μισές αρχές στις οποίες βασίζονται οι έλεγχοι στο διαδικαστικό μοντέλο δεν ισχύουν στο αντικειμενοστρεφές.

Ακολούθως περιγράφουμε αναλυτικά τις διαφορές αυτές, καθώς και τις δυσκολίες με τις οποίες αυτές σχετίζονται.

Figure 1: Τα πιο σχετικά αξιώματα του Hayes

Axiom	**Description**	**Traditional**	**Object-Oriented**
Complexity	For all n, there is a program that is adequately tested by a test set of size n, but not by a test set of size n-1	There is a minimum set of inputs that must be tested	There is a minimum set of inputs and object states that must be tested
Anti-extensionality	There are programs P and Q that compute the same functions (semantically similar), where T is adequate for P but not for Q	It cannot be assumed that the same test cases can be used for different programs that accomplish the same things	It cannot be assumed that the same test cases can be used for functionally similar programs, this can be extended to mean that just because one state is correct for one program, that does not mean that is correct for a similar program
General Multiple Change	There are programs P and Q that are syntactically similar, where T is adequate for P but not for Q	Syntax does not tell you what needs to be tested	The syntax of two programs does not determine the test sets, this also means that if two programs use the same classes, the test cases should be different because the messages sent between them may be different
Anti-decomposition	There is a program P and component C where T is adequate for P and T' is the subset of T that can be used for Q, but T' is not adequate for Q.	A component of a program (say a method) can be adequately tested for use within one program, but not necessarily on its own.	"When a new subclass is added (or an existing subclass is modified) all the methods inherited from each of its ancestor super classes must be retested."
Anti-composition	There exist programs P and Q and a test set T where T is adequate for P and the subset of T that can be used for Q is adequate for Q, but T is not valid for P;Q (the composition of P and Q).	Two programs (or methods) can be adequately tested on their own, but once combined or used in another class; they may no longer be adequately tested.	"If only one module of a program is changed, it seems intuitive that testing should be able to be limited to just the modified unit. However, [this] states that every dependent unit must be retested as well."

2.1 Κατάσταση

Στο διαδικαστικό μοντέλο ένας έλεγχος μπορεί να θεωρηθεί ως ένα διατεταγμένο σύνολο ζευγών εισόδου-εξόδου, αφού το αποτέλεσμα μιας κλήσης στο μοντέλο αυτό εξαρτάται μόνο από την είσοδο που της δίνεται.

Στο αντικειμενοστρεφές μοντέλο κάθε αντικείμενο έχει κατά κανόνα μια εσωτερική κατάσταση. Οι μέθοδοι που καλούνται στο αντικείμενο αυτό όχι μόνο παράγουν εξόδους αλλά και αλλάζουν την εσωτερική του κατάσταση (έχουν δηλαδή παρενέργειες – side effects).

Είναι επομένως εμφανές, πως στο αντικειμενοστραφές μοντέλο η παραπάνω αναπαράσταση του ελέγχου δεν εφαρμόζεται, αφού το αποτέλεσμα μιας κλήσης καθορίζεται από την κατάσταση του αντικειμένου στο οποίο καλείται, η οποία μπορεί να αλλάξει μεταξύ των κλήσεων. Μία επιπλέον δυσκολία είναι ότι η αλλαγή της κατάστασης πολλές φορές δε γίνεται αντιληπτή παρατηρώντας τις εξόδους του προγράμματος.

Για να αντεπεξέλθουν οι έλεγχοι στις νέες απαιτήσεις, πρέπει να λαμβάνεται υπόψιν κατά τον έλεγχο και η αλλαγή στην εσωτερική κατάσταση των αντικειμένων και όλες οι διαφορετικές σειρές κλήσεων των μεθόδων του.

Πιο συγκεκριμένα σύμφωνα με τους Gordon and Roggio [7], για έναν ορθό και ανεπτυγμένο έλεγχο απαιτούνται:

- Μια λίστα από μηνύματα και πράξεις που *μπορούν* να εκτελεστούν από αυτόν.
- Μια λίστα από εξαιρέσεις που μπορεί ή πρέπει να πεταχτούν.
- Ένα σταθερό περιβάλλον εκτέλεσης.
- Όλες οι συμπληρωματικές πληροφορίες που είναι απαραίτητες στον έλεγχο.

2.2 Πολυμορφισμός

Ο πολυμορφισμός εισαγάγει το πρόβλημα ότι δεν υπάρχουν σταθερές υποθέσεις κατά τη διάρκεια της μεταγλώττισης. Επειδή είναι μια δυναμική λειτουργία, είναι αδύνατο να ξέρουμε κατά τον έλεγχο *ποιος* κώδικας θα εκτελεστεί.

Οι κλάσεις στο αντικειμενοστρεφές μοντέλο είναι κατά κανόνα άπειρα επεκτάσιμες, και κάθε επέκταση της κλάσης μπορεί να επεκτείνει και να επαναπροσδιορίσει τις λειτουργίες των κλάσεων πάνω από αυτή στην ιεραρχία. Για κάθε

έλεγχο επομένως θα αναγκαζόμασταν να ελέγξουμε όλες τις πιθανές υποκλάσεις του αντικειμένου, πράγμα αδύνατο, αφού οποτεδήποτε μπορούμε να προσθέσουμε μια καινούρια υποκλάση της κλάσης του.

Ένα καλό παράδειγμα του προβλήματος δίνεται από τους Tauro, Ganesan, and Ghosh [19]:

> “
> Ως παράδειγμα, έστω μια μέθοδος που χρησιμοποιείται για τη σχεδίαση γραφικών. Αν περάσετε τρεις παραμέτρους, η μέθοδος draw δημιουργεί ένα τρίγωνο, αν τέσσερις ένα ορθογώνιο, αν πέντε ένα πεντάγωνο κ.ο.κ. Στην περίπτωση αυτή υπάρχουν τρεις διαφορετικές μέθοδοι με το ίδιο όνομα, αλλά δέχονται διαφορετικό αριθμό παραμέτρων κατά την κλήση τους. Γενικά, η επιλογή της σωστής υπερφορτωμένης μεθόδου καθορίζεται κατά την εκτέλεση (δυναμική δέσμευση) από τον μεταγλωττιστή με βάση τον τύπο και τον αριθμό των ορισμάτων που δίνονται κατά την κλήση:
>
> - Χρειαζόμαστε μόνο μία υπερφορτωμένη μέθοδο;
> - Ελέγχουμε όλες τις υπερφορτωμένες μεθόδους;
> - Αν όλες, χρειάζεται να τις δοκιμάσουμε όλες σε όλες τις κλάσεις της ιεραρχίας;
>
> Δεν υπάρχει μια συγκεκριμένη απάντηση για αυτά τα ερωτήματα. Οι απαντήσεις σε αυτά τα ερωτήματα εξαρτώνται από τους ελεγκτές, τις εταιρικές πολιτικές κ.λπ. Σε έναν τέλειο κόσμο θα δοκιμάζαμε τα πάντα. Ωστόσο, στην πραγματικότητα είναι συχνά αδύνατο να δοκιμάσουμε τα πάντα σε έργα μεγάλης κλίμακας.
> ”

Οι συγγραφείς εδώ αναφέρονται και στην λειτουργία της δυναμικής δέσμευσης (dynamic binding). Για παράδειγμα η C++ χρησιμοποιεί κυρίως στατική δέσμευση, η οποία γίνεται κατά τη διάρκεια της μεταγλώττισης, ενώ η Java δυναμικό, μέσω του JVM (Java Virtual Machine). Στην περίπτωση της δεύτερης,

επειδή το πρόγραμμα ξέρει μόνο κατά τη διάρκεια της εκτέλεσης τους τύπους παραμέτρων, επιστροφής και άλλων δεδομένων που θα χρησιμοποιηθούν, τα παραπάνω προβλήματα διογκώνονται.

Οι συγγραφείς συμβουλεύουν το πρόβλημα του πολυμορφισμού να μετακινηθεί όσο γίνεται στο επίπεδο Ελέγχου Συστήματος και Συνένωσης αντί για το επίπεδο Ελέγχου Μονάδας.

Στο Sharma, Porwal, and Sharma [17] μια λύση που προτείνεται είναι να οριστεί μια προδιαγραφή ιεραρχίας («hierarchy specification»), δηλαδή όλες οι υποκλάσεις συμφωνούν σε μια κοινή, ελάχιστη λειτουργικότητα. Άλλες λύσεις θα αναλυθούν αργότερα στο έγγραφο.

2.3 Κληρονομικότητα

Στο παραπάνω εδάφιο έγινε μια σύντομη αναφορά στο πρόβλημα της επικάλυψης (override) των μεθόδων της υπερκλάσης. Το πρόβλημα αυτό δεν περιορίζεται μόνο στο άπειρο πλήθος των πιθανών αντικειμένων που απαντάνε μια κλήση, αλλά επεκτείνεται και στο πεδίο της ίδιας της υπερκλάσης.

Έστω ότι έχουμε μια πλήρως ελεγμένη κλάση A και μια υποκλάση της B που επαναπροσδιορίζει μόνο ένα υποσύνολο των μεθόδων της A. Θα ήταν φυσικό να υποθέσουμε ότι χρειάζεται να ελέγξουμε μόνο τις μεθόδους αυτές. Σύμφωνα με τους Perry and Kaiser [14] όπως αναφέρεται από τους Barbey and Strohmeier [1], αυτή η υπόθεση δεν ισχύει, επειδή οι αλλαγμένες μέθοδοι ενδέχεται να χρησιμοποιούνται από άλλες κληρονομημένες, αλλάζοντας έτσι τη συμπεριφορά των δεύτερων ή αλλάζοντας την εσωτερική κατάσταση των αντικειμένων. Ένα ακόμα πρόβλημα που δημιουργεί η κληρονομικότητα είναι η παραβίαση της ενθυλάκωσης, καθώς μια υποκλάση μπορεί να έχει πρόσβαση - έμμεσα ή άμεσα – στα δεδομένα των άμεσων και έμμεσων υπερκλάσεών της.

Επομένως, αναγκαζόμαστε είτε να επαληθεύσουμε εκ νέου όλες τις ιδιότητες της καινούριας κλάσης είτε να ανακαλύψουμε το ελάχιστο σύνολο από τις ιδιότητες οι οποίες είναι διαφορετικές από την κλάση-γονέα. Αλγόριθμοι και τεχνικές για την εύρεση αυτού του συνόλου δίνονται αργότερα στο έγγραφο.

Περιέργως, η πολλαπλή κληρονομικότητα δεν δημιουργεί προβλήματα στη διαδικασία της επαλήθευσης, τουλάχιστον όσον αφορά στις μοντέρνες αντικειμενοστρεφείς γλώσσες (Barbey and Strohmeier [1], page 11).

Figure 2: Πιθανές αλλαγές στον κώδικα των ελέγχων

Components		Changes	
Data	Component Changes	1	Change data definition/declaration/uses
		2	Change data access scope/mode
		3	Add/delete data
Method	Interface Changes	4	Add/delete external data usage
		5	Add/delete external data updates
		6	Add/delete/change a method call/a message
		7	Change its signature
	Structure Changes	8	Add/delete a sequential segement
		9	Add/delete/change a branch/loop
	Component Changes	10	Change a control sequence
		11	Add/delete/change local data
		12	Change a sequential segement*
Class	Component Changes	13	Change a defined/redefined method
		14	Add/delete a defined/redefined method
		15	Add/delete/change a defined data attribute
		16	Add/delete a virtual abstract method
		17	Change as attribute access mode/scope
	Relationship Changes	18	Add/delete a superclass
		19	Add/delete a subclass
		20	Add/delete as object pointer
		21	Add/delete an aggregated object
		22	Add/delete an object message
Class	Component Changes	22	Change a class (defined attributes)
Library	Relationship Changes	23	Add/delete a relation between classes
		24	Add/delete a class and its relations
		25	Add/delete an independent class

2.4 Ενθυλάκωση

Η ενθυλάκωση (encapsulation / data hiding) είναι μια θεμελιώδης αρχή του αντικειμενοστραφούς προγραμματισμού, σύμφωνα με την οποία τα εσωτερικά δεδομένα ενός αντικειμένου μπορούν να μεταβληθούν και να προσπελαστούν μόνο μέσω μιας διεπαφής. Εφόσον, όπως εξηγήθηκε παραπάνω, ο έλεγχος της εσωτερικής κατάστασης του αντικειμένου είναι απαραίτητος για τον έλεγχο κλάσεων, ο περιορισμός της ορατότητας δε μας επιτρέπει να αξιολογήσουμε ορθά την κατάσταση αυτή.

Οι διαισθητικές λύσεις απαιτούν είτε να παραβούμε την ενθυλάκωση την ίδια, είτε να παραγάγουμε κλάσεις που μας παρέχουν μεθόδους πρόσβασης. Η πρώτη λύση παραβαίνει τον λόγο ύπαρξης της κλάσης ενώ η δεύτερη ενδέχεται να μην λύσει καν το πρόβλημα, σε περιπτώσεις όπου η υποκλάση δεν έχει πρόσβαση στα εσωτερικά δεδομένα της εξεταζόμενης κλάσης.

Τέλος περιλαμβάνουμε έναν πίνακα (Πίνακας 2.4) από τους Kung et al. [11] στον οποίο συνοψίζονται οι κύριες αλλαγές στους ελέγχους, που επιφέρουν αλλαγές στον κώδικα.

3 Λύσεις

3.1 Έλεγχος Μονάδων

Κατά τον Έλεγχο Μονάδων συγγράφονται περιπτώσεις ελέγχου οι οποίες καλύπτουν κατά το δυνατόν περισσότερες καταστάσεις των αντικειμένων του λογισμικού σε μια προσπάθεια εύρεσης λαθών. Όσο η πολυπλοκότητα του λογισμικού αυξάνεται, ο εξαντλητικός έλεγχος όλων των μονοπατιών εκτέλεσης και όλων των συμπεριφορών των αντικειμένων του συστήματος καθίσταται πρακτικά αδύνατος. Αυτό δυσχεραίνεται από το γεγονός ότι ο αριθμός των πιθανών καταστάσεων ενός μόνο αντικειμένου του λογισμικού είναι απαγορευτικά μεγάλος για να ελεγχθεί με το χέρι και τα εργαλεία αυτόματης κατασκευής περιπτώσεων ελέγχου κατασκευάζουν ένα επίσης απαγορευτικά μεγάλο δέντρο πιθανών καταστάσεων. Ακόμη, ο παραδοσιακός τρόπος ελέγχου με hard-coded δεδομένα αισθητά περιορίζει το πλήθος διαφορετικών μονοπατιών εκτέλεσης που όντως ελέγχονται με αποτέλεσμα μια οριστική απάντηση για το αν υπάρχουν ακόμη σφάλματα να είναι ακόμη πιο δύσκολο να δοθεί.

3.1.1 YAQC4J

Ένα παράδειγμα προσαρμογής πιο παραδοσιακών τεχνικών ελέγχου στο αντικειμενοστραφές μοντέλο είναι το εργαλείο είναι το YAQC4J. Η τεχνική που χρησιμοποιείται είναι η τυχαιοποιημένη παραγωγή δεδομένων, η οποία πρωτοεμφανίστηκε στον συναρτησιακό προγραμματισμό, και συγκεκριμένα με το εργαλείο QuickCheck[15]. Οι βασικές αρχές που διέπουν τη λειτουργία του QuickCheck αναπροσαρμόστηκαν και επεκτάθηκαν για το αντικειμενοστρεφές μοντέλο προγραμματισμού και υλοποιούνται στο εργαλείο YAQC4J, το οποίο μπορεί να χρησιμοποιηθεί συμπληρωματικά με τις υπάρχουσες μεθόδους ελέγχου, την αποτελεσματικότητα των οποίων επαυξάνει[2].

Το εργαλείο αναλαμβάνει να παραγάγει τα τυχαιοποιημένα αντικείμενα και έπειτα να εκτελέσει τις περιπτώσεις ελέγχου έναν μεγάλο αριθμό φορών έως ότου είτε μια εκτέλεση να αποτύχει, είτε ένας προκαθορισμένος αριθμός από εκτελέσεις να επιτύχει. Αν αποτύχει, μαθαίνουμε ότι βρέθηκε κάποιο σφάλμα στο λογισμικό, αλλιώς μαθαίνουμε με μεγάλο αριθμό βεβαιότητας πως το λογισμικό δεν περιέχει σφάλματα.

Παρόλο που βασίζεται στην ίδια λογική με τον προκάτοχό του, το YAQC4J σίγουρα δεν χαίρει την ίδια δημοσιότητα με αυτόν. Αυτό, μαζί με την σχετική έλλειψη δημοφιλών εργαλείων που βασίζονται στην παραγωγή τυχαίων δε-

δομένων, είναι μια ένδειξη ότι η συγκεκριμένη τεχνική **δεν** προτιμάται από τους προγραμματιστές αντικειμενοστραφών προγραμμάτων.

3.2 Έλεγχος Συνένωσης

Όσο προχωράει η ανάπτυξη του λογισμικού, δίδεται όλο και περισσότερη προσοχή στη λειτουργικότητα του λογισμικού, επομένως ο έλεγχος συνένωσης οφείλει να εξασφαλίζει τη σωστή συμπεριφορά του λογισμικού όσο συνενώνει διαφορετικές μονάδες μεταξύ τους. Οι καταστάσεις των αντικειμένων καθώς και η υπερφόρτωση μεθόδων και η δυναμική σύνδεση αντικειμένων με τις αναφορές τους καθιστούν τον έλεγχο διεπαφών δύσκολο, καθώς σε κάθε εκτέλεση και η εσωτερική κατάσταση των αντικειμένων αλλάζει, αλλά και σε κάθε κλήση μεθόδου αντιστοιχούν πιθανόν πολλές υλοποιήσεις. Προβλήματα που πρέπει να αντιμετωπιστούν κατά τον έλεγχο συνένωσης είναι τα εξής:

1. Θέλουμε να έχουμε τον σκελετό του συστήματος το νωρίτερο δυνατό
2. Κατά τη συνένωση προσέχουμε έτσι ώστε οι διεπαφές των μονάδων ενώνονται και αλληλοεπιδρούν σωστά
3. Οι μονάδες επικοινωνούν αρκούντως ικανοποιητικά έτσι ώστε να μπορεί να διεξαχθεί έλεγχος συστήματος.

Προσέγγιση Για τον Έλεγχο Συνένωσης πρέπει να απαντήσουμε στα ακόλουθα τρία ερωτήματα:

1. Πόσα αντικείμενα πρέπει να φτιάξουμε προτού αρχίσουμε τους ελέγχους;
2. Με ποια σειρά πρέπει να συνενώσουμε τις διαφορετικές μονάδες;
3. Χρειάζεται πάνω από ένας σκελετός για την συνένωση;

Η προτεινόμενη μέθοδος που απαντάει στις 3 αυτές ερωτήσεις είναι η ακόλουθη και χρησιμοποιεί μια «εικονική μηχανή συνένωσης» που είναι υπεύθυνη για τη συνένωση των μεμονωμένων κομματιών ενός συστήματος. Η μηχανή έχει ως είσοδο μια λίστα με τα use cases του λογισμικού καθώς και μια λίστα με τα αντικείμενα που θα χρησιμοποιηθούν. Προσπαθεί να εκτελέσει με τη σειρά κάθε use case και κάθε φορά που απαιτείται κάποιο καινούριο αντικείμενο, επιλέγεται κάποιο από τη λίστα, το οποίο ακολούθως ελέγχεται σε συνδυασμό με τα

υπόλοιπα που έχουν προηγουμένως επιλεγεί για σωστή λειτουργικότητα και συνένωση.

Αυτή η μηχανή απαντάει στις τρεις αυτές ερωτήσεις καθώς κατασκευάζει αντικείμενα όταν αυτά χρειαστούν, οι διαφορετικές μονάδες συνενώνονται σύμφωνα με τη σειρά που καθορίζεται από το λογισμικό και παρέχει έναν σκελετό για τη συνένωση.

3.3 Διαφορικός Έλεγχος

Μια από τις μεγαλύτερες δυσκολίες κατά τον έλεγχο είναι η αξιολόγηση του αποτελέσματος του ελέγχου. Τα σύγχρονα συστήματα λογισμικού είναι τόσο περίπλοκα που αποτελεί συχνά πρόκληση ο εκ των προτέρων προσδιορισμός του αναμενόμενου αποτελέσματος, πόσο μάλλον για τους τυχαιοποιημένους ελέγχους. Απαιτείται, λοιπόν, η χρήση ενός «μαντείου» με τη βοήθεια του οποίου να αποφανθούμε αν ένας έλεγχος είναι επιτυχής ή όχι. Τα καταστροφικά λάθη στο λογισμικό εντοπίζονται σχετικά εύκολα, όμως τα λογικά και σημασιολογικά λάθη δύσκολα ανιχνεύονται με παραδοσιακές μεθόδους.

Ο Διαφορικός Έλεγχος αποτελεί προσπάθεια εντοπισμού εκείνων των δυσεύρετων λογικών και σημασιολογικών σφαλμάτων στο λογισμικό δίνοντας την ίδια είσοδο σε πολλές παρόμοιες εφαρμογές και παρατηρώντας διαφοροποιήσεις στην έξοδο του λογισμικού ή στη συμπεριφορά του. Χρησιμοποιώντας σαν «μαντείο» τις διαφορετικές υλοποιήσεις εντοπίζονται αποκλίσεις στη συμπεριφορά των προγραμμάτων οι οποίες αποτελούν πιθανό σφάλμα στο λογισμικό [12].

Μη-Καθοδηγούμενη Παραγωγή Εισόδων Νέες είσοδοι παράγονται τυχαία, αγνοώντας τα αποτελέσματα από τις προηγούμενες εκτελέσεις. Μια τέτοια στρατηγική δεν είναι αποδοτική καθώς οι είσοδοι επιλέγονται από ένα απαγορευτικά μεγάλο σύνολο πιθανών εισόδων. Η πιθανότητα εύρεσης στην τύχη εκείνων των εισόδων που θα εμφανίσουν κάποιο σφάλμα λογισμικού είναι εξαιρετικά μικρή, δεδομένης της λανθάνουσας φύσης των σφαλμάτων.

Καθοδηγούμενη Παραγωγή Εισόδων Λαμβάνοντας υπόψιν πληροφορίες για τη συμπεριφορά του λογισμικού από προηγούμενες εκτελέσεις μπορούμε πιο αποδοτικά να ελέγξουμε το λογισμικό με τις εισόδους εκείνες οι οποίες είναι πιο πιθανό να αποκαλύψουν κάποιο σφάλμα του λογισμικού.

Εξελικτική Καθοδήγηση Ανεξάρτητη από το Πεδίο Χρησιμοποιώντας μετρικές λογισμικού μπορούμε να αξιολογήσουμε τις διαφοροποιήσεις μεταξύ των παρόμοιων προγραμμάτων για να προσδιοριστεί καλύτερα η ποικιλία που υπάρχει μεταξύ τους. Έτσι, επιλέγοντας αυτά τα προγράμματα που εμφανίζουν μεγαλύτερη ποικιλία, εφαρμόζονται πιο αποτελεσματικά τεχνικές διαφορικού ελέγχου με μεθόδους μαύρου κουτιού που είναι ανεξάρτητες πάντα του πεδίου.

Συμβολική Εκτέλεση Αυτή η τεχνική ελέγχου άσπρου-κουτιού εκτελεί το πρόγραμμα συμβολικά, υπολογίζοντας περιορισμούς σε κάθε μονοπάτι εκτέλεσης τους οποίους έπειτα επιλύει προκειμένου να παραγάγει εισόδους που ικανοποιούν αυτούς τους περιορισμούς για κάθε μονοπάτι εκτέλεσης. Αυτή η τεχνική, αν και είναι δύσκολο να χρησιμοποιηθεί σε κλίμακα λόγω της εκθετικής πολυπλοκότητας που προκύπτει από τον αριθμό διαφορετικών μονοπατιών εκτέλεσης στα διάφορα παρόμοια προγράμματα, μπορεί να παραγάγει εισόδους για τον διαφορικό έλεγχο.

3.4 Αρχιτεκτονική

Η ευκολία σύνταξης σωστών ελέγχων εν μέρει εξαρτάται από τη συνοχή της κλάσης στην οποία αυτοί εφαρμόζονται (Yeresime et al. [20] όπως αναφέρεται από τους Gordon and Roggio [7]). Στο διαδικαστικό μοντέλο η συνοχή (cohesion) σχετίζεται με παρόμοιες παραμέτρους και λειτουργικότητα ενώ στο αντικειμενοστραφές με το πόσο συνεκτικές είναι οι επιμέρους κλάσεις του προγράμματος. Μια κλάση θεωρείται συνεκτική αν όλες οι μέθοδοί της την ορίζουν καλά ως μια αυτοτελή μονάδα.

Μια κλάση με υψηλή συνοχή θα έχει μεγάλη ομοιότητα στους τύπους και τις λειτουργίες της, άρα η παραγωγή δεδομένων ελέγχου είναι εύκολη. Στην περίπτωση της χαμηλής συνοχής, η ύπαρξη πολλών ετερογενών δεδομένων και λειτουργιών σημαίνει ότι οι έλεγχοι έχουν μεγαλύτερο κόστος και είναι πιο ευάλωτοι σε σφάλματα.

Οι Gordon and Roggio [7] προτείνουν μια μονάδα μέτρησης η οποία αναπαριστά τη συνοχή μιας κλάσης. Παραθέτουμε τον ορισμό της:

> “
> Έλλειψη συνοχής στις μεθόδους (Lack of Cohesion in Methods - LCOM) ορίζεται ως η μαθηματική διαφορά μεταξύ του αριθμού των μεθόδων των οποίων η μεταβλητές στιγμιοτύπου είναι εντελώς ανόμοιες, και του αριθμό των μεθόδων των οποίων οι μεταβλητές στιγμιοτύπου μοιράζονται (Yeresime, et al., 2012).
> [...]
> Εάν η μετρική LCOM είναι υψηλή, σημαίνει ότι μια κλάση δεν είναι συνεκτική (και μπορεί να είναι υποψήφια για αναδιαμόρφωση σε δύο κλάσεις). Στο στάδιο των ελέγχων, μια κλάση θα πρέπει να έχει διαφορετικά σύνολα ελέγχων για τις διάφορες μεθόδους και όχι ένα σύνολο ελέγχων για ολόκληρη την κλάση. Αυτό οδηγεί σε σύγχυση και αύξηση της συνολικής πολυπλοκότητας της διαδικασίας ελέγχων.» Σύμφωνα με τους Badri and Fadel (Badri & Toure, 2012) η LCOM είναι μια από τις καλύτερες μετρικές για να προβλεφθεί η πολυπλοκότητα του κώδικα.
> ”

Μια δεύτερη έννοια που σχετίζεται με την ευκολία σύνταξης ελέγχων είναι η σύζευξη (coupling). Γενικά υπάρχει μια αντιστρόφως ανάλογη σχέση μεταξύ συνοχής και σύζευξης: όσο πιο πολλή συνοχή έχει μια κλάση τόσο πιο λίγη σύζευξη έχει και αντίστροφα (Khatri et al. [10] όπως αναφέρεται από τους Gordon and Roggio [7]).

Η σύζευξη είναι μια μέτρηση που εκτιμάει τον βαθμό εξάρτησης μεταξύ κλάσεων του προγράμματος. Ενώ λοιπόν η συνοχή ασχολείται με τις σχέσεις εσωτερικά μιας κλάσης, η σύζευξη ασχολείται με τις σχέσεις *μεταξύ* κλάσεων.

Η υψηλή σύζευξη σημαίνει ότι όλες ή οι περισσότερες μέθοδοι πρέπει να κατανοηθούν σαν ένα στενά συνδεδεμένο σύνολο, αντί για ανεξάρτητες λειτουργίες. Αυτό προφανώς επηρεάζει αρνητικά την επαλήθευση αφού οι Έλεγχοι Μονάδας είναι πολύ πιο δύσκολο να γραφτούν. Η χαμηλή σύζευξη είναι ένας από τους στόχους της καλής αρχιτεκτονικής του λογισμικού, ιδιαίτερα για μεγάλα, περίπλοκα συστήματα στα οποία πρέπει κατά το δυνατόν να αποφεύγεται η ανάθεση της επαλήθευσης στον έλεγχο Συστήματος και Συνένωσης.

Η δοκιμή των χαρακτηριστικών θα πρέπει να γίνεται σε Ελέγχους Μονά-

δας και κλάσης. ΄Αλλες δοκιμές περιλαμβάνουν μεθόδους με τιμές επιστροφής που είναι πολυμορφικές, καθώς και πολυμορφικές παραμέτρους. Αυτό γίνεται ευκολότερα σε συστάδες ή σε Ελέγχους Συστήματος.

3.5 Κληρονομικότητα

Η υλοποίηση κλάσεων είναι πιο απλή όταν γίνεται από την κορυφή της ιεραρχίας προς τα κάτω. Κατά τον ίδιο τρόπο, ο έλεγχος των κλάσεων σε μια ιεραρχία κληρονομικότητας είναι γενικά πιο απλός όταν προσεγγίζεται από την κορυφή προς τα κάτω. Εξετάζοντας πρώτα την κορυφή μιας ιεραρχίας, μπορούμε να ασχοληθούμε με την κοινή διεπαφή και τον κώδικα και στη συνέχεια με τον κώδικα της περίπτωσης ελέγχου για κάθε υποκλάση. Η υλοποίηση ιεραρχιών κληρονομικότητας από κάτω προς τα πάνω μπορεί να απαιτήσει σημαντική αναδιαμόρφωση του κοινού κώδικα σε μια νέα υπερκλάση. Οι πιθανότητες αντιμετώπισης διαφορετικών δομών κληρονομικότητας και η πρόσθετη δυνατότητα πιθανής αντιμετώπισης πολλαπλών μορφών κληρονομικότητας μπορεί να προσθέσει άλλο ένα επίπεδο πολυπλοκότητας στη διαδικασία δοκιμής. Τα θέματα αυτά εγείρουν μια σειρά από ερωτήματα:

- Ελέγχουμε πλήρως όλες τις βασικές κλάσεις και τις υποκλάσεις τους και σε ποια επίπεδα πρέπει να ελέγξουμε;
- Δοκιμάζουμε πλήρως όλες τις βασικές κλάσεις ή μόνο τις αλλαγές ή τροποποιήσεις στις υποκλάσεις, και αν ναι σε ποια επίπεδα;
- Με ποια σειρά ελέγχουμε την ιεραρχία, από πάνω προς τα κάτω ή από κάτω προς τα πάνω;

Για να απαντήσουμε στην 3η ερώτηση, η γενικά αποδεκτή απάντηση είναι από πάνω προς τα κάτω. Οι υπόλοιπες ερωτήσεις απαντώνται κυρίως στην ενότητα της Παλινδρόμησης.

3.6 Τεχνητή Νοημοσύνη

Μελετούμε τη συνεισφορά της Τεχνητής Νοημοσύνης στο έλεγχο λογισμικού εξετάζοντας πώς έχει χρησιμοποιηθεί στα διαφορετικά τμήματα του κύκλου ζωής ελέγχου λογισμικού.

3.6.1 Επίδραση της Τεχνητής Νοημοσύνης στον Έλεγχο Λογισμικού

Προδιαγραφή των Ελέγχων [9] Στην αρχή του κύκλου ζωής ελέγχου λογισμικού συγγράφονται περιπτώσεις ελέγχου σύμφωνα με τις απαιτήσεις του λογισμικού. Οι περιπτώσεις ελέγχου οργανώνονται σε ένα έγγραφο προδιαγραφών προκειμένου να εξασφαλιστεί ότι όλες οι απαιτήσεις έχουν ελεγχθεί.

Σε αυτήν τη διαδικασία μπορούν να χρησιμοποιηθούν Info-Fuzzy Networks (IFN) για την εκμαίευση λειτουργικών απαιτήσεων από τα δεδομένα εκτέλεσης με σκοπό την ανάκτηση ελλειπόντων ή ατελών προδιαγραφών, τον προσδιορισμό ενός ελαχίστου συνόλου ελέγχων και την αξιολόγηση της ορθότητας των εξόδων του λογισμικού.

Ξεσκαρτάρισμα Περιπτώσεων Ελέγχου Είναι η διαδικασία κατά την οποία επιλέγονται για εκτέλεση μόνο οι πιο αποδοτικές περιπτώσεις ελέγχου με αποτέλεσμα τη μείωση του συνολικού κόστους του ελέγχου.

Χρησιμοποιώντας Info-Fuzzy Networks (INF) εντοπίζονται αυτόματα συσχετίσεις μεταξύ εισόδων και των αντίστοιχων εξόδων του λογισμικού. Το INF έπειτα μαθαίνει από αυτές τις συσχετίσεις προκειμένου να δημιουργήσει ένα δέντρο κατηγοριοποίησης για τις περιπτώσεις ελέγχου από το οποίο μετά επιλέγονται οι πιο σημαντικές. Έτσι μειώνεται σημαντικά ο αριθμός συνδυαστικών ελέγχων μαύρου κουτιού.

Παραγωγή Περιπτώσεων Ελέγχου Τον προσδιορισμό κριτηρίων καταλληλότητας ελέγχου ακολουθεί η παραγωγή ενός συνόλου ελέγχων που συμμορφώνεται με αυτά τα κριτήρια. Μια τέτοια διαδικασία είναι εμφανώς υπερβολικά περίπλοκη για να εκτελεστεί με το χέρι, ειδικά για πολύπλοκα συστήματα, επομένως υπάρχει μια στροφή προς στην τεχνητή νοημοσύνη σχετικά με την αυτόματη παραγωγή των περιπτώσεων ελέγχου.

Αρχικά χρησιμοποιήθηκαν μέθοδοι επαγωγικής μάθησης για την παραγωγή από παραδείγματα εισόδων και εξόδων εκείνων των περιπτώσεων ελέγχου οι οποίες αρκούν για τη διαφοροποίηση του λογισμικού P που ελέγχεται από κάθε άλλο λογισμικό P' ενός συνόλου εναλλακτικών λογισμικών.

Για τον έλεγχο μαύρου κουτιού κατασκευάζεται ένα μοντέλο της συμπεριφοράς του λογισμικού μέσω παραδειγμάτων εισόδων-εξόδων, το οποίο χρησιμοποιείται για να δημιουργήσει νέες εισόδους.

Περιπτώσεις ελέγχου μπορούν να παραχθούν αναλύοντας απευθείας το Έγγραφο Προσδιορισμού Απαιτήσεων Λογισμικού χρησιμοποιώντας τεχνικές επεξεργασίας φυσικής γλώσσας (Natural Language Processing, NLP).

Παραγωγή Δεδομένων Ελέγχου Για την εκτέλεση των Περιπτώσεων Ελέγχου απαιτείται παραγωγή των δεδομένων τα οποία θα χρησιμοποιηθούν για τον έλεγχο του λογισμικού. Όσο καλύτερη είναι η ποιότητα αυτών των δεδομένων τόσο μεγαλύτερη θα είναι και η κάλυψη του κώδικα από τις περιπτώσεις ελέγχου.

Με τη βοήθεια γενετικών αλγορίθμων πραγματοποιείται αναζήτηση στον χώρο των εισόδων του λογισμικού για την εύρεση δεδομένων τα οποία καλύπτουν κάθε μονοπάτι εκτέλεσης.

Χρησιμοποιώντας βαθιά μάθηση, με τη μέθοδο της «μαϊμούς» γίνεται εκμάθηση των εισόδων που θα έβαζε ο ελεγκτής του λογισμικού και στατιστική συσχέτισή τους με τα συμφραζόμενα. Ακολούθως η «μαϊμού» προβλέπει νέες εισόδους με βάση τα παρατηρούμενα συμφραζόμενα σε κάθε περίπτωση.

Κατασκευή «Μαντείου» (Oracle) Ελέγχου Προκειμένου να επιτελέσει ο έλεγχος λογισμικού σωστά τη λειτουργεία του χρειάζεται ένας μηχανισμός ο οποίος να έχει τη δυνατότητα, δεδομένης μιας εισόδου στο λογισμικό, να διαχωρίσει τη σωστή και λανθάνουσα συμπεριφορά του λογισμικού. Αυτός ο μηχανισμός αποτελεί το πρόβλημα του μαντείου (oracle problem).

Αλγόριθμοι μηχανικής μάθησης μπορούν να χρησιμοποιηθούν για την αυτόματη κατασκευή μαντείων, ακόμα και σε περιπτώσεις όπου οι προδιαγραφές λογισμικού είναι ελλιπείς. Κάθε εκτέλεση του ελέγχου παράγει αποτελέσματα με τα οποία τροφοδοτείται ο αλγόριθμος. Το μοντέλο που προκύπτει χρησιμοποιείται ως το μαντείο.

Όταν υπάρχει ένα μοντέλο αναφοράς του λογισμικού, τότε μπορεί επίσης να χρησιμοποιηθεί επιβλεπόμενη μηχανική μάθηση μέσω της οποίας το μοντέλο που προκύπτει μαθαίνει να διαχωρίσει τις περιπτώσεις ελέγχου ως επιτυχείς και ανεπιτυχείς.

Η επιβλεπόμενη μάθηση εφαρμόζεται και σε νευρωνικά δίκτυα, όπου το μοντέλο μαθαίνει να διαχωρίζει μοτίβα εκτέλεσης για επιτυχείς και μη εκτελέσεις για ένα δοσμένο λογισμικό. Ένα μικρό υποσύνολο των ιχνών εκτέλεσης κατηγοριοποιούνται ως επιτυχείς ή μη και το μοντέλο έπειτα μαθαίνει από αυτά.

Ιεράρχηση Περιπτώσεων Ελέγχου Οι διάφορες περιπτώσεις ελέγχου μπορούν να εκτελεστούν με πολλούς διαφορετικούς τρόπους. Προσπαθούμε να βρούμε τη βέλτιστη σειρά εκτέλεσης έτσι ώστε να δοθεί προτεραιότητα σε αυτές τις περιπτώσεις ελέγχου οι οποίες είναι πιο πιθανό να αποκαλύψουν ατέλειες του λογισμικού, ή σε αυτές οι οποίες συσχετίζονται με μεγαλύτερο κίνδυνο ανάλογα με, φερ' ειπείν, τη σοβαρότητα του αντικειμένου υπό έλεγχο ή την επίπτωση του

κινδύνου εάν εμφανιστεί.

Ένα σύνολο υπάρχουσων τεχνικών συνενώνεται και λειτουργεί ως μηχανική μάθηση για την ιεράρχηση περιπτώσεων ελέγχου χρησιμοποιώντας πληροφορίες όπως κάλυψη του κώδικα, ηλικία του ελέγχου, ιστορικό αποτυχιών και ομοιότητα του κειμενικού περιεχομένου, προκειμένου να κατασκευάσει ένα μοντέλο που ιεραρχεί αποτελεσματικά περιπτώσεις ελέγχου

Για την Κατάταξη SVM (support-vector machines, επίσης ως και support-vector networks) χρησιμοποιούνται μετά-δεδομένα μαύρου κουτιού, όπως ιστορικό περιπτώσεων ελέγχου, αλλά και οι περιγραφές των περιπτώσεων ελέγχου σε φυσική γλώσσα, προκειμένου να ιεραρχηθούν οι περιπτώσεις χρήσης.

Εκτίμηση του Κόστους των Περιπτώσεων Ελέγχου Η εκτίμηση του κόστους είναι η διαδικασία κατά την οποία προβλέπεται η απαιτούμενη προσπάθεια για την ανάπτυξη του λογισμικού. Γενικά δε θα έπρεπε να υπάρχει έλλειμα στην εκτίμηση, και αυτή θα πρέπει να είναι διαθέσιμη το νωρίτερο δυνατό.

Μοντελοποιώντας το κόστος ως ένα τρισδιάστατο διάνυσμα του αριθμού των περιπτώσεων ελέγχου, της πολυπλοκότητας εκτέλεσης του ελέγχου και των ανθρώπων που ελέγχουν το λογισμικό, δημιουργείται μια βάση με προηγούμενα δεδομένα πάνω στην οποία εφαρμόζεται SVM για την εκτίμηση του κόστους για δεδομένα ιστορικά δεδομένα και διανύσματα που μοντελοποιούν το κόστος.

Η μηχανική μάθηση μπορεί να προβλέψει το μέγεθος του κώδικα ελέγχου, δηλαδή τη μετρική λογισμικού «γραμμές κώδικα ελέγχου», που αποτελεί σημαντικό δείκτη της προσπάθειας που θα καταβληθεί κατά τον έλεγχο. Χρησιμοποιώντας τεχνικές όπως γραμμική παλινδρόμηση, K-κοντινότεροι-γείτονες, αφελής ταξινομητής Bayes, τυχαίο δάσος και Perceptron πολλαπλών επιπέδων κατασκευάστηκε μοντέλο που με ακρίβεια προβλέπει αυτήν τη μετρική.

Επειδή η υποεκτίμηση του κόστους έχει σοβαρές συνέπειες στην ποιότητα του λογισμικού, τα μοντέλα που εκτιμούν το κόστος περιλαμβάνουν κάποια προκατάληψη προς την υπερεκτίμηση. Ακόμη, εντοπίζονται τα χαρακτηριστικά εκείνα που είναι πιο σημαντικά για την πρόβλεψη του κόστους ελέγχου λογισμικού, και συγκεκριμένα του χρόνου ελέγχου.

3.6.2 Γενετικοί Αλγόριθμοι

Όπως και στο διαδικαστικό μοντέλο, έτσι και στο αντικειμενοστρεφές, ο έλεγχος περιπτώσεων δοκιμών μεγάλου μεγέθους θα μπορούσε να καταστήσει ανέφικτη την εκτέλεση όλων των πιθανών περιπτώσεων ελέγχου για την επαλήθευση

Software Testing Activity	**AI Technique Applied**
Test Case Generation	Inductive Learning - Active Learning - Ant colony Optimization - Markov Model - AI Planner -GA - Tabu Search - NLP - Re-enforcement Learning C4.5 - Goal Based - Decision Tree - K-Nearest Neighbour - Logistic Regression - Random Forest - Multi-Layer Perceptron - K star - LSTM - Heuristic Search
Test Data Generation	GA - Simulated Annealing - Hill Climbing - Generative Model - LSTM - Deep Re-enforcement Learning - Ant Colony Optimization - Heuristic Methods
Test Oracle Construction	ANN - SVM - Decision Trees - AdaBoostM1 - Incremental Reduced Error Pruning (IREP) - Info Fuzzy Network
Test Case Prioritization	K-Means - Expectation-Maximization - c4.5 - Cob Web - Reinforcement Learning - CBR - ANN - Markov Model - K-NN - Logistic Regression - SVM Rank
Test Case Specification	IFN - C4.5
Test Case Refinement	IFN - Classification Tree Method
Test Cost Estimation	SVM - linear regression - k-NN - Naïve Bayes - C4.5 - Random Forest - Multilayer Perceptron

Table 1: Τεχνικές που εφαρμόζονται ανά τύπο δραστηριότητας ελέγχου

του προγράμματος. Μια λύση σε αυτό το πρόβλημα είναι ο τυχαίος έλεγχος. Οι Ciupa et al. [5] παρέχουν αποδείξεις ότι ο σχετικός αριθμός σφαλμάτων που ανιχνεύονται από τυχαίο έλεγχο με την πάροδο του χρόνου είναι προβλέψιμος και μάλιστα ότι αυτός βρίσκει σφάλματα σχετικά γρήγορα. Η πρώτη αποτυχία είναι πιθανό να προκληθεί εντός 30 δευτερολέπτων. Οι εξελικτικοί αλγόριθμοι, όπως είναι οι γενετικοί αλγόριθμοι, έχουν χρησιμοποιηθεί ευρέως για διαδικαστικούς ελέγχους λογισμικού. Σε ορισμένα μέρη, προσεγγίσεις που βασίζονται σε γενετικούς αλγορίθμους έχουν προταθεί για τη δοκιμή αντικειμενοστρεφούς λογισμικού.

Μια διαισθητική χρήση γενετικού προγραμματισμού είναι να κατασκευαστούν έλεγχοι οι οποίοι αποτελούνται από μια ακολουθία κλήσεων μεθόδων. Αυτό θα μπορούσε να υλοποιηθεί κωδικοποιώντας μεθόδους σαν αναγνωριστικά τα οποία ο αλγόριθμος θα αξιολογούσε χρησιμοποιώντας μία συνάρτηση καταλληλότητας (fitness function). Παρόλα αυτά αποδείχτηκε, πως η υλοποίηση αυτή παράγει τόσες πολλές άκυρες ακολουθίες κλήσεων που ο αλγόριθμος δεν είναι αποδοτικός.

Στο Meziane and Vadera [13] γίνεται αναφορά στους S. and J. [16]), οι οποίοι χρησιμοποίησαν επίσης γενετικό προγραμματισμό, όχι για τυχαιοποιημένο έλεγχο, αλλά για να παραχθούν αυτόματοι έλεγχοι με την παραγωγή και εξέλιξη δέντρων τα οποία αναπαριστούν συναρτήσεις ή μεθόδους. Η ακριβής υλοποίηση δίνεται παρακάτω:

> “
> Η βασική αναπαράσταση στα περισσότερα συστήματα γενετικού προγραμματισμού είναι ένα δέντρο αντί της αριθμημένης λίστας. Γενικά, ένα δέντρο αναπαριστά μια συνάρτηση, όπου οι κόμβοι-φύλλα αναπαριστούν τα ορίσματα και οι ενδιάμεσοι κόμβοι τις συναρτήσεις. Στο πλαίσιο των ελέγχων, τα δέντρα αυτά μπορούν να αναπαραστήσουν τις εξαρτήσεις μεταξύ των κλήσεων μεθόδων, οι οποίες μπορούν στη συνέχεια να μετατραπούν σε γραμμική μορφή για την παραγωγή δοκιμών. Η χρήση της μετάλλαξης και της διασταύρωσης σε αυτά τα δέντρα μπορεί να οδηγήσει σε άκυρες συναρτήσεις και ακατάλληλα ορίσματα στο πλαίσιο της δοκιμής αντικειμενοστρεφών προγραμμάτων.
> ”

> “
> Ως εκ τούτου, οι Wappler και Wegener(2006), προτείνουν τη χρήση ισχυρά τυποποιημένου γενετικού προγραμματισμού (Montana, 1995), όπου οι τύποι των κόμβων χρησιμοποιούνται για να διασφαλιστεί ότι εξελίσσονται μόνο δέντρα με τις κατάλληλες παραμέτρους. Αυτό εξακολουθεί να αφήνει ανοιχτό το ζήτημα του τρόπου με τον οποίο φτιάχνονται αυτά τα δέντρα εξ αρχής. Η προσέγγιση που υϊοθετούν είναι να ληφθεί πρώτα ένας γράφος εξάρτησης κλήσεων μεθόδων που έχει ακμές μεταξύ των κόμβων των κλάσεων και των κόμβων των μεθόδων. Οι ακμές καθορίζουν τις μεθόδους που μπορούν να χρησιμοποιηθούν για τη δημιουργία στιγμιοτύπων μιας κλάσης, και τα στιγμιότυπα χρειάζονται από μια συγκεκριμένη μέθοδο. Αυτός ο γράφος μπορεί στη συνέχεια να διανυθεί για τη δημιουργία δέντρων που θα αποτελούν τον αρχικό πληθυσμό. Τα απαιτούμενα ορίσματα (αντικείμενα) για τα δέντρα λαμβάνονται από μια διαδικασία δευτέρου επιπέδου που περιλαμβάνει αρχικά τη δημιουργία γραμμικών ακολουθιών κλήσης μεθόδων από τα δέντρα και στη συνέχεια χρησιμοποιεί έναν γενετικό αλγόριθμο για την εύρεση των πιο κατάλληλων δέντρων. Μόλις επιτευχθεί αυτό, τα δέντρα βελτιστοποιούνται με τη χρήση επανασυνδυασμού και μεταλλάξεων με γνώμονα ορισμένους στόχους, όπως η κάλυψη των μεθόδων.
> ”

Μια σημαντική βελτίωση στο έργο των Wappler and Wegener έφερε ο Ribeiro (2008), ο οποίος εκτέλεσε τον παραπάνω αλγόριθμο αγνοώντας μεθόδους που δεν είχαν καμία παρενέργεια (δηλαδή δεν μετέβαλλαν την εσωτερική κατάσταση του αντικειμένου). Μια δοκιμή στην κλάση Stack της Java Standard Library απέδειξε ότι αυτή η βελτίωση μείωσε το χρόνο εκτέλεσης έως την πλήρη κάλυψη κατά δύο τρίτα.

Οι Meziane and Vadera [13] αναφέρουν και ακόμη μια ενδιαφέρουσα χρήση γενετικών αλγορίθμων (ΓΑ) στο πεδίο της ανάπτυξης ελέγχων:

Οι C., Feng, and Y. [4] διερευνούν τη χρήση ΓΑ για τον προσδιορισμό της

βέλτιστης σειράς για την ενσωμάτωση και τον έλεγχο κλάσεων. Ένα σημαντικό πρόβλημα κατά τον προσδιορισμό της κατάλληλης σειράς εμφανίζεται επειδή πρέπει να σπάσουν οι κύκλοι εξάρτησης των κλάσεων καθιστώντας απαραίτητη τη χρήση στελεχών (stubs). Η πολυπλοκότητα των απαιτούμενων στελεχών ποικίλλει, ανάλογα με το επίπεδο σύζευξης που υπάρχει μεταξύ των κλάσεων, συνεπώς διαφορετικές διατάξεις απαιτούν διαφορετικά επίπεδα προσπάθειας για τη δημιουργία των στελεχών.

Οι C., Feng, and Y. [4] εκμεταλλεύονται προηγούμενες εργασίες για τον προγραμματισμό, για παράδειγμα στη χρήση ΓΑ για το πρόβλημα του πλανόδιου πωλητή, και χρησιμοποιούν μια κωδικοποίηση των κλάσεων με μεταθέσεις μαζί με μια συνάρτηση καταλληλότητας που μετρά το βαθμό σύζευξης μεταξύ των κλάσεων. Το μέτρο καταλληλότητας ορίζεται έτσι ώστε να μειώνεται ο αριθμός των χαρακτηριστικών, οι μέθοδοι που θα χρειαζόταν να αλλάξουν αν αποκοπεί μια εξάρτηση και ο αριθμός των στελεχών που δημιουργούνται. Επιπλέον, δεν επιτρέπουν την αποκοπή των εξαρτήσεων κληρονομικότητας και σύνθεσης, καθώς οδηγούν σε δαπανηρά στελέχη. Έπειτα από πειραματισμό με αυτή την προσέγγιση σε μια μελέτη περίπτωσης ATM χρησιμοποιώντας το σύστημα ΓΑ "Evolver" (Palisade, 1998), η σύγκριση των αποτελεσμάτων με εκείνα που προέκυψαν με τη χρήση μιας προσέγγισης βασισμένης σε γράφους και προκύπτει το συμπέρασμα ότι η χρήση ΓΑ μπορεί να παρέχει μια καλύτερη προσέγγιση για την παραγωγή διατάξεων ενσωμάτωσης και δοκιμής κλάσεων.

3.6.3 Νευρωνικά δίκτυα

Στο Sharma, Porwal, and Sharma [17] γίνεται αναφορά στο έργο των Gong and Li [6], οι οποίοι πρότειναν την χρήση Event-Driven Petri Nets (EDPN) έτσι ώστε να μοντελοποιηθούν οι μεταβολές στην εσωτερική κατάσταση και στην συμπεριφορά των αντικειμένων.

> “
> Τα σφάλματα ανιχνεύονται με την ανάλυση των διαφορών των σεναρίων ελέγχου στις δυναμικές συμπεριφορές των EDPN και η μέθοδος μπορεί να επιλέξει μια περίπτωση ελέγχου που ανιχνεύει σφάλματα που περιγράφονται στα μοντέλα σφαλμάτων. Οι Ghang et al. πρότειναν μια μέθοδο που χρησιμοποιεί ελέγχους σχετικής αξιοπιστίας και πρόβλεψη της αξιοπιστίας των μονοπατιών εκτέλεσης για την προσαρμογή της κατανομής των δοκιμών αξιοπιστίας λογισμικού στον αντικειμενοστρεφή προγραμματισμό. Στην προσέγγισή τους, ο έλεγχος αξιοπιστίας λογισμικού βασίζεται στα προφίλ λειτουργίας και στην πρόβλεψη της σχετικής αξιοπιστίας των μονοπατιών λειτουργίας. Για το σκοπό αυτό χρησιμοποίησαν έναν αλγόριθμο μάθησης νευρωνικού δικτύου.
> ”

Μέχρι στιγμής το έγγραφο αυτό έχει επικεντρωθεί αποκλειστικά στην επαλήθευση του λογισμικού, εφόσον αυτό είναι το θέμα το οποίο καταναλώνει τον περισσότερο χρόνο και είναι πιο άμεσα σχετικό με το αντικειμενοστρεφές μοντέλο. Παρόλα αυτά η επικύρωση του λογισμικού, ο έλεγχος ότι η λειτουργικότητα του λογισμικού ανταποκρίνεται στις απαιτήσεις του πελάτη, είναι ένα εξίσου σημαντικό κομμάτι της διαδικασίας ανάπτυξης. Εδώ τα νευρωνικά δίκτυα μπορούν να μας βοηθήσουν αναλύοντας τις απαιτήσεις του πελάτη, οι οποίες είναι γραμμένες σε φυσική γλώσσα, και μετατρέποντάς τες σε μορφή κατάλληλη για τους σχεδιαστές και τους προγραμματιστές, αποφεύγοντας έτσι ανθρώπινα λάθη στη διαδικασία αυτή.

Οι Batarseh et al. [3] περιγράφουν ένα τέτοιο σύστημα:

“
Οι περιπτώσεις χρήσης που προέκυψαν ήταν σαφείς και συνεπείς, ανεξάρτητα από το μέγεθος του κειμένου των απαιτήσεων. Τα R-Tool, NL-OOPS, CM-BUILDER είναι μερικά ακόμη εργαλεία μηχανικής λογισμικού υποβοηθούμενης από υπολογιστή, που βασίζονται σε NLP. Τα εργαλεία αυτά παράγουν διαγράμματα κλάσεων από το έγγραφο απαιτήσεων χρήστη (αν και εξακολουθεί να απαιτεί την παρέμβαση του χρήστη).
Οι Michl et al. πρότειναν το NL-OOPS (Natural Language - Object-Oriented Production System) για την ανάπτυξη ενός εργαλείου που υποστηρίζει αντικειμενοστρεφή ανάλυση. Τα έγγραφα απαιτήσεων αναλύονται με το LOLITA (Large-scale Object-based Linguistic Interactor, Translator and Analyzer), ένα σύστημα επεξεργασίας φυσικής γλώσσας μεγάλης κλίμακας. Και οι γνώσεις που περιέχονται στα έγγραφα αλλά και αυτές που είναι ήδη αποθηκευμένες στη Βάση Γνώσης του LOLITA προτείνονται στη συνέχεια για χρήση για την παραγωγή μοντέλων απαιτήσεων. Η προσέγγιση ήταν βασισμένη στη θεώρηση ότι οι απαιτήσεις συχνά γράφονται σε αδόμητη φυσική γλώσσα και σε πολλές περιπτώσεις είναι αδύνατο να επιβληθούν περιορισμοί στη γλώσσα που χρησιμοποιείται για τη συγγραφή τους.
Η αντικειμενοστρεφής μονάδα μοντελοποίησης υλοποίησε έναν αλγόριθμο που φιλτράρει τους κόμβους οντοτήτων και συμβάντων στη βάση γνώσης του για εντοπισμό κλάσεων και συσχετίσεων. Παράλληλα, οι Ninaus et al. πρότειναν μια μέθοδο για τη μείωση του κινδύνου των απαιτήσεων χαμηλής ποιότητας μέσω της βελτίωσης της υποστήριξης των ενδιαφερόμενων μερών στην ανάπτυξη μοντέλων RE. Οι συγγραφείς εισήγαγαν το περιβάλλον INTELLIREQ, το οποίο βασίζεται σε διαφορετικές προσεγγίσεις προτάσων, οι οποίες προτάσεις υποστηρίζουν τα ενδιαφερόμενα μέρη στις δραστηριότητες που σχετίζονται με τις απαιτήσεις, όπως ο ορισμός, η διασφάλιση της ποιότητας, η επαναχρησιμοποίηση και ο τελικός σχεδιασμός.
”

3.6.4 Δοκιμές Παλινδρόμησης

Σύμφωνα με τους Kung et al. [11], ο έλεγχος παλινδρόμησης πρέπει να αντιμετωπίσει τέσσερα θεμελιώδη προβλήματα:

- Πώς να προσδιορίσουμε αυτόματα τα στοιχεία εκείνα που έχουν επηρεαστεί λόγω αλλαγής ορισμένων άλλων στοιχείων;
- Ποια στρατηγική πρέπει να χρησιμοποιηθεί για την επαναληπτική δοκιμή αυτών των επηρεασμένων στοιχείων;
- Ποια είναι τα κριτήρια κάλυψης για τον επανέλεγχο αυτών των στοιχείων;
- Πώς να επιλέξουμε, να επαναχρησιμοποιήσουμε και να τροποποιήσουμε τον υπάρχοντα έλεγχο (και να δημιουργηθούν νέοι);

Λύσεις σε αυτά τα προβλήματα για τα παραδοσιακά προγράμματα έχουν προταθεί τις τελευταίες δύο δεκαετίες. Ωστόσο, στον έλεγχο των αντικειμενοστρεφών προγραμμάτων δεν έχει δοθεί πολλή προσοχή, ενώ στην παλινδρόμηση σχεδόν καθόλου. Αυτό γίνεται για τρείς κύριους λόγους, σύμφωνα με τον συγγραφέα:

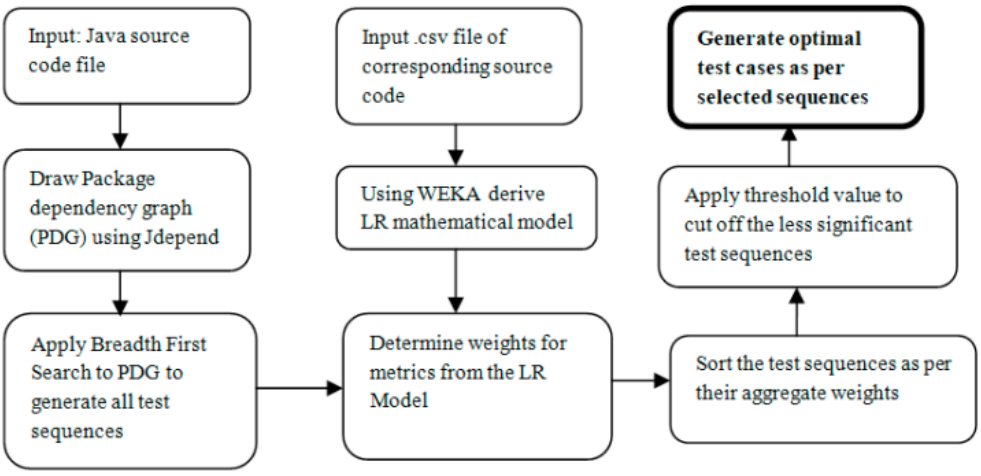
Πρώτον, οι παραδοσιακές προσεγγίσεις δεν αντιμετωπίζουν τις πολύπλοκες σχέσεις και εξαρτήσεις, όπως η κληρονομικότητα, η συνάθροιση και η συσχέτιση, που υπάρχουν μεταξύ των κλάσεων. Δεύτερον, οι περισσότερες παραδοσιακές προσεγγίσεις βασίζονται στο μοντέλο ροής ελέγχου, αλλά τα αντικείμενα των κλάσεων έχουν συμπεριφορά εξαρτώμενη από κατάσταση που μπορεί να αλλάξει με διάφορους τρόπους. Ως εκ τούτου, οι παραδοσιακές προσεγγίσεις δεν μπορούν να εφαρμοστούν στον έλεγχο κλάσεων. Τρίτον, οι παραδοσιακές προσεγγίσεις χρησιμοποιούν στελέχη για να προσομοιώσουν τις μονάδες που καλούνται, αλλά σε αντικειμενοστρεφή προγράμματα αυτό είναι δύσκολο και δαπανηρό, επειδή απαιτεί την κατανόηση πολλών σχετικών κλάσεων, των μεθόδων τους και του τρόπου με τον οποίο καλούνται οι μέθοδοι.

3.6.5 Έλεγχοι Παλινδρόμησης

Στο Tanejaa et al. [18] χρησιμοποιείται γραμμική παλινδρόμηση σε αρχεία πηγαίου κώδικα java για να παραχθεί το ελάχιστο σύνολο ελέγχων που πρέπει να εκτελεστούν ώστε να αντιμετωπιστεί η μεγάλη πλειοψηφία πιθανών σφαλμάτων.

Ο αλγόριθμος εξετάζει πολλές μετρικές αξιολόγησης του λογισμικού, σε μερικές από τις οποίες έχουμε ήδη αναφερθεί (σαν το LCOM) και αξιολογεί

Figure 3: Το πλάνο εκτέλεσης του προγράμματος των Tanejaa et al. [18]



πόσο «συνεισφέρει» κάθε μονάδα στην πιθανότητα κάποιου bug. Το λογισμικό WAKA παράγει μια πρωτοβάθμια εξίσωση στην οποία κάθε μετρική είναι μια παράμετρος. Αντικαθιστώντας κάθε παράμετρο με τη μετρική του κάθε λογισμικού παράγεται ένας αριθμός που αξιολογεί την καταλληλότητα ενός ελέγχου να βρίσκει σφάλματα.

Τέλος, ταξινομούμε κάθε ακολουθία από ελέγχους με βάση την παραπάνω τιμή και φτιάχνουμε έναν πίνακα (Πίνακας 3.6.5) με τους πιο αποτελεσματικούς ελέγχους. Στο παρακάτω παράδειγμα αυτές θα ήταν οι TS3, TS4, TS5, TS6, TS7, TS10.

Οι συγγραφείς καταλήγουν στο ότι «μπορεί εύκολα να διαπιστωθεί ότι σχεδόν το 80% των σφαλμάτων έχουν καλυφθεί στο 60% του συνολικού χρόνου. Επομένως, συνάγεται το συμπέρασμα ότι ο αριθμός των σφαλμάτων που αποκαλύπτονται ανά μονάδα χρόνου είναι καλύτερος με την προτεινόμενη μεθοδολογία».

Για να εκτιμήσουμε τα πλεονεκτήματα αυτής της μεθόδου και το πώς η τεχνητή νοημοσύνη τα ενισχύει, μπορούμε να την συγκρίνουμε με μια πιο παραδοσιακή μέθοδο.

Οι Kung et al. [11] οι οποίοι χρησιμοποιούν μαθηματικές λύσεις, κυρίως βασισμένες σε θεωρία γράφων, ώστε να κατασκευάσουν αλγορίθμους που εντοπίζουν σχέσεις εξάρτησης μεταξύ στοιχείων του προγράμματος και άρα ποιοι έλεγχοι πρέπει να μεταβληθούν. Η θεωρία και οι αλγόριθμοι που χρησιμοποιούνται καθώς και οι μέθοδοι αξιοποίησής τους διαφεύγουν της εμβέλειας αυτού του

Figure 4: Τα αποτελέσματα του προγράμματος για κάθε ακολουθία ελέγχου

Test Sequences ID	Nodes Covered	Weight
TS_1	1,6	4.7438
TS_2	1,5	4.4612
TS_3	1,4	10.0074
TS_4	2,1	10.6780
TS_5	2,1,4	**16.3882**
TS_6	2,1,5	10.8420
TS_7	2,1,6	12.2246
TS_8	3,5	1.4982
TS_9	4,5	5.8742
TS_{10}	4,1,6	10.5540

εγγράφου και άρα παραπέμπουμε τον αναγνώστη να διαβάσει την αναφορά ο ίδιος.

Γίνεται σαφές ότι αυτή η προσέγγιση, αν και θεωρητικά πιο αξιόπιστη, υστερεί σε δυνατότητα (και ευκολία) αυτοματοποίησης με την εναλλακτική παραπάνω λύση. Αυτός είναι πιθανώς και ο λόγος που η συγκεκριμένη έρευνα τελικά απέτυχε να παράξει πρακτικό αποτέλεσμα στο αντικειμενοστραφές πεδίο.

3.7 Αυτοματοποίηση

Μακράν ο πιο συνηθισμένος και συχνά πιο πρακτικός τρόπος να διευκολύνουμε την επαλήθευση αντικειμενοστρεφών προγραμμάτων παραμένει η αυτοματοποίηση των ελέγχων. Αυτή είναι απαραίτητη για τη παράδοση προϊόντος με καλή ποιότητα στον κατάλληλο χρόνο.

Η αυτοματοποίηση δοκιμών είναι λογισμικό που αυτοματοποιεί οποιαδήποτε πτυχή της δοκιμής ενός συστήματος εφαρμογών με δυνατότητα παραγωγής δοκιμαστικών εισόδων και αναμενόμενων αποτελεσμάτων. Μειώνει τις επαναλαμβανόμενες εργασίες που κάνουμε χειροκίνητα και επίσης μας παρέχει το αποτέλεσμα ”επιτυχία” ή ”αποτυχία”.

Σύμφωνα με Tauro, Ganesan, and Ghosh [19], η κατάλληλη έκταση των αυτοματοποιημένων δοκιμών εξαρτάται από

> “
> [...] τους στόχους των ελέγχων, τον προϋπολογισμό, τη διαδικασία λογισμικού, το είδος του υπό ανάπτυξη λογισμικού και τις ιδιαιτερότητες του περιβάλλοντος ανάπτυξης και του περιβάλλοντος στο οποίο θα τρέχει το λογισμικό τελικά. Οι αυτοματοποιημένοι έλεγχοι εξασφαλίζουν χαμηλό ποσοστό ελαττωμάτων και συνεχή πρόοδο, ενώ οι χειροκίνητοι έλεγχοι θα οδηγούσαν πολύ γρήγορα σε εξάντληση των ελεγκτών.
> ”

Προσθέτοντας αργότερα πως:

“

Για να συνοψίσουμε τα χαρακτηριστικά των ελέγχων που επιδιώκουμε:

- Οι έλεγχοι *εκτελούν* το σύστημα σε αντίθεση με τις στατικές αναλύσεις που αναλύουν απλά τον κώδικα.
- Οι έλεγχοι είναι αυτόματοι για να αποφευχθούν περιπτώσεις όπου τα μέλη του έργου να λόγω οκνηρίας δεν πραγματοποιούν τους ελέγχους (ή εναλλακτικά για να αποφευχθούν περιπτώσεις όπου ένα σύστημα που δεν έχει ελεγχθεί αρκετά).
- Οι αυτοματοποιημένοι έλεγχοι δημιουργούν τα δεδομένα ελέγχου, εκτελούν τον έλεγχο και εξετάζουν το αποτέλεσμα αυτόματα.
- Η επιτυχία ή η αποτυχία του ελέγχου παρατηρείται αυτόματα κατά τη διάρκεια της εκτέλεσης του ελέγχου.
- Μια σουίτα ελέγχων ορίζει επίσης ένα σύστημα που εκτελείται μαζί με το δοκιμασμένο σύστημα παραγωγής. Ο σκοπός αυτού του εκτεταμένου συστήματος είναι να εκτελεί τους ελέγχους σε αυτοματοποιημένη μορφή.
- Ένας έλεγχος είναι υπόδειγμα. Ένας έλεγχος χρησιμοποιεί συγκεκριμένες τιμές για τα δεδομένα εισόδου, τα δεδομένα ελέγχου.
- Ένας έλεγχος είναι επαναλήψιμος και καθορισμένος. Για τις ίδιες παραμέτρους παράγονται τα ίδια αποτελέσματα.

”

Το σημείο στο οποίο ξεχωρίζει όμως η αυτοματοποίηση, και ο λόγος που

χρησιμοποιείται ευρύτατα (σχεδόν καθολικά) στην επαλήθευση λογισμικού είναι ότι μέσω αυτής μπορούν να χρησιμοποιηθούν όλες σχεδόν οι τεχνικές στις οποίες αναφερθήκαμε σε αυτό το έγγραφο - εφόσον αυτές δεν είναι υπολογιστικά δύσκολες. Παραδοσιακοί έλεγχοι μονάδας, έλεγχοι τυχαίων δεδομένων και έλεγχοι παλινδρόμησης μπορούν εύκολα να εκτελούνται ανά τακτά χρονικά διαστήματα, χωρίς την παρέμβαση του προγραμματιστή.

4 Συμπέρασμα

Η σύγχρονη περιόδος, κατα την οποία η ανάπτυξη λογισμικού καθοδειγείται από το αντικειμενοστραφές μοντέλο, έχει οδηγήσει στην αλλαγή τόσο των τεχνικών όσο και των εργαλείων που χρησιμοποιούνται για την επαλήθευσή του. Τεχνικές που βασίζονται στην παραγωγή δεδομένων ή στην μαθηματική επαλήθευση των δοκιμασιών επιλέγονται σπάνια, αντίθετα με άλλα μοντέλα ανάπτυξης όπως το συναρτησιακό, ενώ απλές τεχνικές όπως η κατασκευή ελέγχων μονάδας χρησιμοποιούνται εκτενώς με την βοήθεια εργαλείων αυτοματοποίησης. Η έρευνα στον τομέα αυτόν επίσης δείχνει προς την παραγωγή σωστών ελέγχων μέσω σωστού σχεδιασμού και αρχιτεκτονικής, αντί για την εύρεση κάποιου βέλτιστου τρόπου επαλήθευσης. Σε κάθε περίπτωση, λόγω της ραγδιαίας εξέλιξης του πεδίου, της σχετικά νεογνούσας έρευνας γύρω από αυτό και την συνεχή ανάπτυξη άλλων εργαλείων (π.χ Τεχνητής Νοημοσύνης) ενδέχεται να ανακαλυφθούν στο προσεχές μέλλον καινούριοι τρόποι και εργαλεία που οδηγούν σε πιο αποτελεσματική και εύχρηστη παραγωγή ελέγχων.

References

[1] S. Barbey and A. Strohmeier. “The problematics of testing object-oriented software”. In: *Swiss Federal Institute of Technology, Computer Science Department, Software Engineering Laboratory, EPFL-DI-LGL* (1994).

[2] Pablo Andrés Barrientos. “An Approach for Unit Testing in OOP Based on Random Object Generation”. In: *Computational Science and Its Applications* (2014). DOI: https://doi.org/10.1007/978-3-319-09156-32.

[3] Feras A. Batarseh et al. “Chapter 10: The Application of Artificial Intelligence in Software Engineering – A Review Challenging Conventional Wisdom”. In: *Nexus of Artificial Intelligence, Software Development, and Knowledge Engineering* (2021). URL: https://arxiv.org/abs/2108.01591.

[4] Briand L. C., J. Feng, and Labiche Y. “Using genetic algorithms and coupling measures to devise optimal integration test orders”. In: *Proceedings of the Fourteenth International Conference on Software Engineering and Knowledge Engineering* (2002).

[5] I. Ciupa et al. “On the number and nature of faults found by random testing.” In: *Special Issue:ICST 2008, the First IEEE International Conference on Software Testing, Verification and Validation* 21 (1 2011). DOI: https://doi.org/10.1002/stvr.415.

[6] H. Gong and J. Li. ““Generating Test Cases of ObjectOriented Software Based on EDPN and Its Mutant”. In: *9th International Conference for Young Computer Scientists* (2008).

[7] Jamie S. Gordon and Robert F. Roggio. “A Comparison of Software Testing Using the Object-Oriented Paradigm and Traditional Testing”. In: *Proceedings of the Conference for Information Systems Applied Research San Antonio, Texas, USA* 67 (2013).

[8] Hayes and Jane Huffman. “Testing of Object Oriented Programming Systems (OOPS): A Fault-Based Approach”. In: *Notes in Computer Science* 858 (1994).

[9] Zubair Khaliq, Sheikh Umar Farooq, and Dawood Ashraf Khan. “Artificial Intelligence in Software Testing : Impact, Problems, Challenges and Prospect”. In: *University of Kashmir* (2022). DOI: https://doi.org/10.48550/arXiv.2201.05371.

[10] Khatri et al. “Analysis Of Factors Affecting Testing In Object-oriented Systems”. In: *International Journal On Computer Science And Engineering 3* (2011).

[11] David C. Kung et al. “On Regression Testing of Object-Oriented Programs”. In: *Journal of Systems and Software* 32 (1 1996).

[12] William M. McKeeman. “Differential Testing for Software”. In: *DIGITAL TECHNICAL JOURNAL* 10.1 (1998), pp. 100–107.

[13] Farid Meziane and Sunil Vadera. *Artificial Intelligence Applications for Improved Software Engineering Development: New Prospects*. 2009. DOI: 10.4018/978-1-60566-758-4.ch014. URL: http://usir.salford.ac.uk/id/eprint/2208/.

[14] Dewayne E. Perry and Gail E. Kaiser. "Adequate testing and object-oriented programming." In: *Journal of Object-Oriented Programming* (1990).

[15] *QuickCheck official page*. https://hackage.haskell.org/package/QuickCheck.

[16] Wappler S. and Wegener J. "Evolutionary unit testing of object-oriented software using strongly-typed genetic programming." In: *Proceedings of the Eighth Annual Conference on Genetic and Evolutionary Computation* (2006).

[17] Chandra Mani Sharma, Rabins Porwal, and Deepika Sharma. "Testing Object Oriented Software: Issues, State-of-the-art and Future". In: *Journal of Systems and Software* 32 (2013).

[18] Divya Tanejaa et al. "A Novel technique for test case minimization in object oriented testing." In: *International Conference on Computational Intelligence and Data Science* (2019).

[19] Clarence Tauro, Nagesswary Ganesan, and Anupam Ghosh. "Testing Object-Oriented Software Systems: A Survey of Steps and Challenges". In: *International Journal of Computer Applications* 42 (2012).

[20] Yeresime et al. "Effectiveness of Software Metrics For Object-Oriented System". In: *Procedia Technology, 2nd International Conference on Communication, Computing and Security* 6 (n.d).