Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων σε Συστήματα με Κατάσταση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ ΔΟΝΤΑΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2021

Εθνικό Μετσόβιο Πολυτεχνείο
Σχολή Ηλεκτρολόγων Μηχανικών
και Μηχανικών Υπολογιστών
Τομέας Τεχνολογίας Πληροφορικής και
Υπολογιστών

Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων σε Συστήματα με Κατάσταση

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΣΠΥΡΙΔΩΝ ΔΟΝΤΑΣ

Επιβλέπων : Κωνσταντίνος Σαγώνας
Αν. Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Ιανουαρίου 2021.

....	....	....
Κωνσταντίνος Σαγώνας	Νικόλαος Σ. Παπασπύρου	Γεώργιος Γκούμας
Αν. Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Αν. Καθηγητής Ε.Μ.Π.

Αθήνα, Ιανουάριος 2021

....

Σπυρίδων Δοντάς

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Σπυρίδων Δοντάς, 2021.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

// Περίληψη

Ο έλεγχος βάσει ιδιοτήτων, ή Property Based Testing, αποτελεί μία τεχνική η οποία χρησιμοποιείται για τον έλεγχο είτε αγνών συνάρτησεων είτε περίπλοκων συστημάτων με εσωτερική κατάσταση. Το PropEr, το οποίο είναι το εργαλείο που χρησιμοποιείται στο πλαίσιο αυτής της διπλωματικής, παρέχει, επιπλέον, τη δυνατότητα στοχευμένου ελέγχου βάσει ιδιοτήτων, ή Targeted Property Based Testing. Αυτός αποτελεί μια παραλλαγή του ελέγχου βάσει ιδιοτήτων όπου χρησιμοποιείται κάποια στρατηγική αναζήτησης για την καθοδήγηση της τυχαίας παραγωγής εισόδων προς κάποιο στόχο, ο οποίος ωστόσο περιορίζεται σε αγνές συναρτήσεις. Στην παρούσα διπλωματική εργασία, επεκτείνουμε το PropEr, και τις ήδη υπάρχουσες υλοποιήσεις για έλεγχο συστημάτων με εσωτερική κατάσταση (`proper_statem` και `proper_fsm`), ώστε να μπορεί να ελέγχει στοχευμένα τέτοια συστήματα. Ο χρήστης το μόνο που καλείται να κάνει είναι να χρησιμοποιήσει το παρεχόμενο API, καθώς και να προσδιορίσει μία τιμή χρησιμότητας προς μεγιστοποίηση ή ελαχιστοποίηση, με το PropEr να αναλαμβάνει όλη την υπόλοιπη διαδικασία. Δεδομένης της τιμής χρησιμότητας, το PropEr θέτει σε λειτουργία στρατηγικές αναζήτησης, οι οποίες, ύστερα από κάποιο αριθμό δοκιμών, πλησιάζουν αυτό το μέγιστο ή ελάχιστο, ώστε να επιβεβαιώσει πως το σύστημα δεν παρουσιάζει σφάλμα, όταν η εσωτερική του κατάσταση έχει κάποια συγκεκριμένη μορφή (από την οποία προκύπτει και η τιμή χρησιμότητας).

Λέξεις κλειδιά

έλεγχος λογισμικού, έλεγχος βάσει ιδιοτήτων, αυτόματος έλεγχος, τυχαίος έλεγχος, στοχευμένος έλεγχος βάσει ιδιοτήτων.

Abstract

Property Based Testing is a widely used technique in order to test either pure functions or complex stateful systems. PropEr, which is the tool that is being used as part of this thesis, provides, in addition, the ability of Targeted Property Based Testing. This is a variation of Property Based Testing in that it employs search strategies to guide the random generation of inputs to a certain target, but is limited to pure functions. In this thesis, we extend PropEr, and the libraries it provides to perform stateful property based testing (`proper_statem` and `proper_fsm`), so that it can perform targeted testing on such systems. The user is only required to use the provided API calls, as well as define a utility value to maximize or minimize. Given that utility value, PropEr will use search strategies, which, after some number of tests, eventually come close, or achieve, this maximum or minimum, so that it can confirm that the system does not produce errors, when its internal state is in some specific format (from which the utility value is derived).

Key words

software testing, property based testing, automated testing, random testing, targeted property based testing.

Ευχαριστίες

Ευχαριστώ θερμά τον επιβλέποντα καθηγητή αυτής της διατριβής, κ. Κωστή Σαγώνα, για την έμπνευση του θέματος αυτής της διπλωματικής, τη συνεχή καθοδήγησή του καθώς και τη συνεργασία μας τα τελευταία δύο χρόνια. Επιπλέον, θα ήθελα να τον ευχαριστήσω για την εμπιστοσύνη του να μου αναθέσει το θέμα της διπλωματικής παρόλο που δεν είχαμε συνεργαστεί στο παρελθόν.

Ευχαριστώ τους καθηγητές και τους φοιτητές του Εργαστηρίου Λογισμικού, καθώς και όλους τους υπόλοιπους καθηγητές που ήταν πρόθυμοι να βοηθήσουν και να συζητήσουν απορίες με εποικοδομητικές συζητήσεις. Θα ήθελα, επίσης, να ευχαριστήσω ιδιαιτέρως τον κ. Νίκο Παπασπύρου, ο οποίος ήταν πηγή έμπνευσης για εμένα και ο λόγος για τον οποίο ασχολούμαι επαγγελματικά με τον προγραμματισμό σήμερα. Η θέλησή του για βοήθεια, συμβουλή και συζήτηση με έκαναν να συνειδητοποιήσω την αγάπη μου για την Πληροφορική.

Τέλος, ευχαριστώ την οικογένειά μου που στάθηκε στο πλάι μου όλα τα χρόνια σε κάθε μου επιλογή, καθώς και τους φίλους και συμφοιτητές μου για τη συνεχή υποστήριξη και για όλα αυτά τα φοιτητικά χρόνια.

Σπυρίδων Δοντάς,

Αθήνα, 22η Ιανουαρίου 2021

Η εργασία αυτή είναι επίσης διαθέσιμη ως Τεχνική Αναφορά CSD-SW-TR-7-20, Εθνικό Μετσόβιο Πολυτεχνείο, Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών, Τομέας Τεχνολογίας Πληροφορικής και Υπολογιστών, Εργαστήριο Τεχνολογίας Λογισμικού, Ιανουάριος 2021.

URL: `http://www.softlab.ntua.gr/techrep/`
FTP: `ftp://ftp.softlab.ntua.gr/pub/techrep/`

Περιεχόμενα

Περίληψη . . . 5

Abstract . . . 7

Ευχαριστίες . . . 9

Περιεχόμενα . . . 11

Κατάλογος σχημάτων . . . 13

Κατάλογος προγραμμάτων . . . 15

1. Εισαγωγή . . . 17
- 1.1 Η Γλώσσα Προγραμματισμού Erlang . . . 17
- 1.2 Δομή της Εργασίας . . . 18

2. Έλεγχος Βάσει Ιδιοτήτων . . . 19
- 2.1 Έλεγχος Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων . . . 21
- 2.2 Ένα Κατατοπιστικό Παράδειγμα . . . 21

3. Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων . . . 27
- 3.1 Προσομοίωση Ανόπτησης . . . 29
- 3.2 Λαβύρινθος . . . 29

4. Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων σε Συστήματα με Κατάσταση . . . 35
- 4.1 Αλγόριθμος . . . 37
- 4.2 Υλοποίηση . . . 38

5. Πειραματική Αξιολόγηση . . . 43
- 5.1 Συχνότητα Εμφάνισης Εντολών . . . 43
- 5.2 Πιθανότητα Εύρεσης Σφάλματος . . . 46
- 5.3 Συμπεράσματα . . . 56

6. Επίλογος . . . 57
- 6.1 Παρόμοια Έργα . . . 57
- 6.2 Μελλοντική Δουλειά . . . 57

Βιβλιογραφία . . . 59

Κατάλογος σχημάτων

2.1 Οι φάσεις του Ελέγχου Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων . . . 22
3.1 Στοχευμένος Έλεγχος Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων 28
3.2 Εύκολος Λαβύρινθος 30
3.3 Δύσκολος Λαβύρινθος 31
4.1 Ποσοστό Εμφάνισης Εντολών 37
5.1 Ποσοστό Εμφάνισης Εντολών στον Τυχαίο Έλεγχο 44
5.2 Ποσοστό Εμφάνισης Εντολών στον Στοχευμένο Έλεγχο 44
5.3 Κατανομή Ποσοστού Εμφάνισης Εντολών στον Τυχαίο Έλεγχο 45
5.4 Κατανομή Ποσοστού Εμφάνισης Εντολών στον Στοχευμένο Έλεγχο . . . 46
5.5 Πιθανότητα Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο 47
5.6 Πιθανότητα Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο 47
5.7 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (100 Δοκιμές) 49
5.8 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (200 Δοκιμές) 49
5.9 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (500 Δοκιμές) 50
5.10 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (1000 Δοκιμές) . . . 50
5.11 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (2000 Δοκιμές) . . . 51
5.12 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (5000 Δοκιμές) . . . 51
5.13 Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (10000 Δοκιμές) . . 52
5.14 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (100 Δοκιμές) . 52
5.15 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (200 Δοκιμές) . 53
5.16 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (500 Δοκιμές) . 53
5.17 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (1000 Δοκιμές) 54
5.18 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (2000 Δοκιμές) 54
5.19 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (5000 Δοκιμές) 55
5.20 Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (10000 Δοκιμές) 55

Κατάλογος προγραμμάτων

2.1 Ιδιότητα Διαγραφής Στοιχείου από Λίστα 20
2.2 Απεικόνιση Κατάστασης 23
2.3 Συνάρτηση επανάκλησης initial_state 23
2.4 Συνάρτηση επανάκλησης command 23
2.5 Γεννήτρια Κλειδιών 23
2.6 Συνάρτηση Επανάκλησης next_state 24
2.7 Ιδιότητα Προσωρινής Μνήμης 24
2.8 Προσυνθήκες 25
2.9 Μετασυνθήκες 25
3.1 Στοχευμένη Ιδιότητα 27
3.2 Συνάρτηση Παραγωγής Μονοπατιών 30
3.3 Ιδιότητα Λαβυρίνθου 30
3.4 Συνάρτηση Γειτνίασης Μονοπατιών 31
3.5 Στοχευμένη Ιδιότητα Λαβυρίνθου 32
4.1 Στοχευμένος Έλεγχος Προσωρινής Μνήμης 35
4.2 Συνάρτηση Παραγωγής commands_gen/1 38
4.3 Συνάρτηση Παραγωγής commands_gen/4 39
4.4 Συνάρτηση Παραγωγής commands/1 39
4.5 Συνάρτηση Παραγωγής next_commands_gen/1 40
4.6 Συνάρτηση Παραγωγής next_commands_gen/5 41
4.7 Δημόσιο API proper_statem 41
4.8 Δημόσιο API proper_fsm 42
4.9 Συνάρτηση Παραγωγής next_commands_gen/2 42

Κεφάλαιο 1

Εισαγωγή

Ο έλεγχος λογισμικού είναι μια τεχνική που χρησιμοποιείται συχνά στον τομέα της Πληροφορικής, που αποσκοπεί στην εύρεση σφαλμάτων του συστήματος υπό έλεγχο, ή να επιβεβαιώσει τη σωστή λειτουργία του. Στο κλασικό μοντέλο προγραμματισμού, ο έλεγχος εγκυρότητας ενός προγράμματος, μιας συνάρτησης ή ακόμα κι ενός συστήματος γίνεται από τον ίδιο τον προγραμματιστή, ο οποίος είναι υπεύθυνος να γράψει μία σειρά από εισόδους και αναμενόμενες εξόδους αυτού. Ύστερα, εκτελεί το πρόγραμμα, τη συνάρτηση ή το σύστημα και παρατηρεί αν υπήρχαν λάθη και αν ναι με ποιες εισόδους προκλήθηκαν. Ωστόσο, η παραπάνω μέθοδος, παρ'όλο που είναι απλή και εύκολη να αυτοματοποιηθεί με εργαλεία μοναδιαίου ελέγχου (στην Erlang για παράδειγμα με το εργαλείο EUnit [1]), παρουσιάζει δυσκολίες. Αρχικά, η συγγραφή ελέγχων είναι δύσκολη και κουραστική όταν γίνεται με το χέρι και, επιπλέον, είναι δύσκολο να καλυφθεί το πλήρες εύρος περιπτώσεων εκτέλεσης του λογισμικού.

Τα τελευταία χρόνια, υπάρχει ένα αυξανόμενο ενδιαφέρον στον έλεγχο λογισμικού βάσει ιδιοτήτων ή Property Based Testing. Είναι μία τεχνική ελέγχου όπου ο έλεγχος βασίζεται σε ιδιότητες του συστήματος υπό έλεγχο, ενώ οι είσοδοι παράγονται από τυχαίες συναρτήσεις παραγωγής. Στην παρούσα διπλωματική, το εργαλείο που χρησιμοποιούμε ονομάζεται PropEr και παρέχει στους χρήστες του τη δυνατότητα να γράφουν τέτοιου είδους ελέγχους, είτε για αγνές συναρτήσεις, είτε για συστήματα με εσωτερική κατάσταση. Πέραν αυτού, το PropEr δίνει τη δυνατότητα για ένα νέο είδος ελέγχου, τον στοχευμένο έλεγχο βάσει ιδιοτήτων, ή Targeted Property Based Testing, μία παραλλαγή του τυχαίου ελέγχου βάσει ιδιοτήτων, ο οποίος, όμως, πριν από την εργασία αυτή μπορούσε να εφαρμοσθεί μόνο για αγνές συναρτήσεις.

Σκοπός της παρούσας εργασίας είναι η επέκταση του εργαλείου PropEr και της δυνατότητάς του για στοχευμένο έλεγχο βάσει ιδιοτήτων, ώστε να μπορεί να ελέγχει με αυτό τον τρόπο και συστήματα με εσωτερική κατάσταση, καθώς και μία αρχική αξιολόγηση αυτής της επέκτασης.

1.1 Η Γλώσσα Προγραμματισμού Erlang

Η Erlang είναι μια γλώσσα προγραμματισμού που χρησιμοποιείται κυρίως στον τομέα των τηλεπικοινωνιακών συστημάτων, τα οποία χρειάζονται υψηλή διαθεσιμότητα. Αναπτύχθηκε από την Ericsson τις δεκαετίες του 1980-1990 και είχε ως σκοπό την ανάπτυξη κλιμακούμενων και εύρωστων συστημάτων πραγματικού χρόνου που λειτουργούν συνεχώς. Τέτοια συστήματα συνήθως αποτελούνται από ένα πλήθος μικρότερων συστημάτων που αλληλεπιδρούν με το περιβάλλον τους βάσει του actor model. Αυτό σημαίνει ότι τα συστήματα επικοινωνούν μεταξύ τους ασύχρονα μέσω μηνυμάτων, γεγονός που διευκολύ-

[1] http://erlang.org/doc/apps/eunit/chapter.html

νει τον έλεγχο συστημάτων που τρέχουν ταυτόχρονα. Αυτή η αλληλεπίδραση δημιουργεί την εσωτερική κατάσταση κάθε επιμέρους συστήματος, και βασίζεται στις προηγούμενες ενέργειες ενώ επηρεάζει και τις επόμενες, εισάγοντας με αυτό τον τρόπο side-effects.

Ο προγραμματισμός για συστήματα που τρέχουν ταυτόχρονα είναι τόσο διαδεδομένος στην Erlang που η ίδια η γλώσσα παρέχει ένα πλήθος βιβλιοθηκών κάτω από την πλατφόρμα Erlang/OTP, η οποία αποτελείται από το runtime σύστημα της Erlang, καθώς και ένα πλήθος από έτοιμα για χρήση components και ένα σετ από σχεδιαστικές επιλογές για προγράμματα σε Erlang. Αυτό που μας ενδιαφέρει, στο πλαίσιο της παρούσας διπλωματικής, είναι οι "εφαρμογές" που παρέχει το OTP για την συγγραφή λογισμικού που αντιπροσωπεύει συστήματα όπως τα παραπάνω. Πιο συγκεκριμένα, παρέχει δύο διεπαφές με τις οποίες ο χρήστης μπορεί να δημιουργήσει συστήματα διεργασιών με κατάσταση, και ονομάζονται `gen_server`[2] και `gen_statem`[3]. Η πρώτη δίνει τη δυνατότητα στο χρήστη να υλοποιήσει ένα server με εσωτερική κατάσταση, ο οποίος λαμβάνει μηνύματα και απαντάει σε αυτά. Η δεύτερη χρησιμεύει στην υλοποίηση ενός state machine, με το οποίο και πάλι ο χρήστης αλληλεπιδρά μέσω μηνυμάτων.

Το PropEr παρέχει δύο βιβλιοθήκες, τις `proper_statem` και `proper_fsm`, οι οποίες χρησιμεύουν στη μοντελοποίηση των παραπάνω εφαρμογών αντιστοίχως, καθώς και στον αυτόματο έλεγχό τους. Οι ονομασίες στο PropEr οφείλονται στο γεγονός ότι όταν υλοποιήθηκαν τα παραπάνω, η βιβλιοθήκη για τα FSMs ονομαζόταν `gen_fsm`. Με αυτή τη λογική το PropEr ονόμασε `proper_statem` τη γενική κατηγορία των συστημάτων με κατάσταση και `proper_fsm` την κατηγορία των FSM. Αυτές είναι οι δύο βιβλιοθήκες που θα επεκταθούν σε αυτή τη διπλωματική με σκοπό, εκτός από τυχαίο έλεγχο, να μπορεί να εκτελεσθεί και στοχευμένος έλεγχος βάσει ιδιοτήτων.

1.2 Δομή της Εργασίας

Η δομή της παρούσας εργασίας έχει ως εξής. Στο κεφάλαιο 2 υπάρχει μια εισαγωγή στον έλεγχο βάσει ιδιοτήτων, τόσο για αγνές συναρτήσεις όσο και για συστήματα με εσωτερική κατάσταση, με τη βοήθεια παραδειγμάτων. Στο κεφάλαιο 3, εξηγούμε τον τρόπο με τον οποίο λειτουργεί ο στοχευμένος έλεγχος βάσει ιδιοτήτων. Το κεφάλαιο 4 είναι το κύριο κεφάλαιο της διπλωματικής, στο οποίο παρουσιάζουμε την υλοποίησή μας καθώς και κάποια στατιστικά με τη βοήθεια ενός παραδείγματος. Το κεφάλαιο 5 συμπληρώνει με διαγράμματα και αξιολόγηση της υλοποίησής μας. Τέλος, η διπλωματική κλείνει με το κεφάλαιο 6, όπου παραθέτουμε τα συμπεράσματά μας, παρόμοιες εργασίες καθώς και την επικείμενη δουλειά πάνω στο αντικείμενο.

[2] https://erlang.org/doc/man/gen_server.html
[3] https://erlang.org/doc/man/gen_statem.html

Κεφάλαιο 2

Έλεγχος Βάσει Ιδιοτήτων

Ο έλεγχος βάσει ιδιοτήτων, ή property based testing (PBT), είναι μία τεχνική ελέγχου στην οποία η προβλεπόμενη συμπεριφορά του συστήματος το οποίο ελέγχεται, ή system under test (SUT), εκφράζεται με ιδιότητες, οι οποίες αναμένεται να κρατήσουν είτε για ολόκληρο το σύστημα είτε για τις μονάδες από τις οποίες αποτελείται. Για κάθε ιδιότητα, ένα εργαλείο ελέγχου βάσει ιδιοτήτων παράγει διαδοχικά τυχαίες εισόδους με αυξανόμενη πολυπλοκότητα (π.χ. μέγεθος εισόδου, πυκνότητα εμφάνισης κάποιας συγκεκριμένης τιμής, κλπ). Στη συνέχεια, το εργαλείο υποβάλλει το σύστημα σε αυτές τις εισόδους και ελέγχει αν οι έξοδοι παραποιούν τις ιδιότητες ή όχι. Ακολουθώντας αυτή τη μεθοδολογία, οι χειροκίνητες εργασίες ενός ελεγκτή μειώνονται στον καθορισμό των παραμέτρων του συστήματος και στη διαμόρφωση ενός συνόλου ιδιοτήτων που περιγράφουν με ακρίβεια την επιδιωκόμενη συμπεριφορά του.

Οι *ιδιότητες* που ελέγχονται είναι ουσιαστικά μια μερική προδιαγραφή του συστήματος, κάτι το οποίο σημαίνει ότι είναι πιο συμπαγείς και ευκολότερες στη συγγραφή και κατανόηση σε σχέση με τις ολοκληρωμένες προδιαγραφές του συστήματος. Οι χρήστες μπορούν να εκμεταλλευθούν πλήρως τη γλώσσα εγγραφής κατά τη σύνταξη ιδιοτήτων, και έτσι μπορούν να περιγράψουν με ακρίβεια ένα μεγάλο εύρος σχέσεων εισόδων-εξόδων. Σε σύγκριση με τον έλεγχο συστημάτων με περιπτώσεις ελέγχου γραμμένες χειροκίνητα, ο έλεγχος με ιδιότητες είναι μία γρηγορότερη και λιγότερο τετριμμένη διαδικασία. Οι προκύπτουσες ιδιότητες είναι επίσης πολύ πιο συμπαγείς από μία μακρά σειρά ελέγχων μονάδων (unit tests), αλλά, όταν χρησιμοποιηθούν σωστά, μπορούν να επιτύχουν πιο εξονυχιστικό έλεγχο του συστήματος μέσω της υποβολής του σε μία αρκετά μεγαλύτερη ποικιλία εισόδων σε σχέση με οτιδήποτε θα ήταν πρόθυμος ή θα μπορούσε να γράψει ένας άνθρωπος. Επιπροσθέτως, οι ιδιότητες μπορούν να χρησιμεύσουν ως μία μερική προδιαγραφή του συστήματος η οποία μπορεί να ελέγχεται, και η οποία είναι γενικότερη από οποιοδήποτε σύνολο ελέγχων μονάδων, και έτσι είναι πολύ καλύτερη στην εξερεύνηση μεγαλύτερων ποσοστών των συμπεριφορών ενός συστήματος και στην αποκάλυψη σφαλμάτων του.

Στα εργαλεία ελέγχου ιδιοτήτων, η διαδικασία ελέγχου μπορεί συνήθως να διαμορφωθεί με διάφορους τρόπους μέσω επιλογών. Για παράδειγμα, οι χρήστες μπορούν να ελέγξουν τον αριθμό των ελέγχων προς εκτέλεση, το μέγεθος των παραγόμενων εισόδων, κλπ.

Ας παρουσιάσουμε τον έλεγχο βάσει ιδιοτήτων με ένα παράδειγμα που χρησιμοποιείται συνήθως: έλεγχος συνάρτησης διαγραφής στοιχείου από λίστα. Μία μετασυνθήκη (postcondition) που περιμένουμε να ισχύει μετά τη διαγραφή κάποιου στοιχείου από τη λίστα είναι ότι το στοιχείο δεν υπάρχει πλέον στη λίστα που προκύπτει. Στη γλώσσα του PropEr [11], το εργαλείο που χρησιμοποιούμε σε αυτή την εργασία, αυτή η ιδιότητα γράφεται όπως φαίνεται στο πρόγραμμα 2.1.

Αυτή η ιδιότητα δηλώνει ότι, για οποιονδήποτε ακέραιο αριθμό `I` και λίστα ακεραίων

```erlang
prop_lists_delete_deletes_all_elements() ->
  ?FORALL({I, L}, {integer(), list(integer())},
          not lists:member(I, lists:delete(I, L))).
```

Πρόγραμμα 2.1: Ιδιότητα Διαγραφής Στοιχείου από Λίστα

`L`, εάν διαγράψουμε το `I` από το `L`, τότε το `I` δε θα αποτελεί μέλος της προκείπτουσας λίστας. Εδώ χρησιμοποιούμε τη λειτουργία `delete` από τη στάνταρ βιβλιοθήκη `lists` ως σύστημα υπό εξέταση. Οι ιδιότητες ξεκινούν με το `prop_` ενώ οτιδήποτε ξεκινάει με αγγλικό ερωτηματικό (?) είναι μία μακροεντολή που αντιστοιχεί σε κάποια δομή που παρέχει το PropEr, και οτιδήποτε ξεκινάει με κεφαλαίο γράμμα (όπως το `L` εδώ) είναι μία μεταβλητή. Εκτός από τις μακροεντολές όπως το `?FORALL`, τα εργαλεία ελέγχου βάσει ιδιοτήτων παρέχουν και ενσωματωμένες γεννήτριες για τους βασικούς τύπους της γλώσσας, όπως οι `integer()` και `list(T)` που χρησιμοποιούνται εδώ.

Ο έλεγχος αυτής της ιδιότητας με το PropEr, αποκαλύπτει γρήγορα ότι η ιδιότητα δεν ισχύει για τη συνάρτηση `delete` της στάνταρ βιβλιοθήκης της Erlang, η οποία αφαιρεί μόνο την πρώτη εμφάνιση του στοιχείου στη λίστα:

```
1> proper:quickcheck(example:prop_lists_delete_deletes_all_elements()).
.............................!
Failed: After 30 test(s).
{2,[-8,2,-16,111,2,2,5,-18,-1]}

Shrinking .....(5 time(s))
{2,[2,2]}
false
```

Βλέπουμε ότι το PropEr έτρεξε 29 δοκιμές που πέρασαν, πριν τη δημιουργία ενός ζεύγους εισόδων για τις οποίες η ιδιότητα απέτυχε. Στη συνέχεια συρρίκνωσε μία από αυτές τις εισόδους (τη λίστα `L`) για να παρουσιάσει στο χρήστη ένα ζεύγος εισόδων ελάχιστου μεγέθους για το οποίο δεν ισχύει η ιδιότητα. Αναμφισβήτητα, η συρρίκνωση είναι ένα από τα πιο χρήσιμα χαρακτηριστικά του ελέγχου βάσει ιδιοτήτων: οδηγεί σε αντιπαραδείγματα όπου *όλα τους τα μέρη* σχετίζονται με την αποτυχία της ιδιότητας.

Παρόλο που αυτό το παράδειγμα είναι πολύ απλό, μας επιτρέπει να κάνουμε δύο παρατηρήσεις που σχετίζονται με το υπόλοιπο αυτής της εργασίας.

1. Σε εργαλεία ελέγχου βάσει ιδιοτήτων, η δημιουργία εισόδων δεν είναι πραγματικά τυχαία: απλά σκεφτείτε την πιθανότητα δημιουργίας του ίδιου ακεραίου τρεις φορές σε ένα ζεύγος ακεραίου-λίστας, όπου η λίστα έχει επίσης μικρό μήκος. Αντ'αυτού, αυτά τα εργαλεία χρησιμοποιούν ενσωματωμένες γεννήτριες που έχουν την τάση να δημιουργούν "ενδιαφέρουσες" μικρές τιμές πρώτα, αυξάνοντας σταδιακά το μέγεθός τους καθώς αυξάνεται ο αριθμός των δοκιμών. Εναλλακτικά, ο χρήστης μπορεί να επηρεάσει άμεσα το μέγεθος των παραγόμενων εισόδων χρησιμοποιώντας επιλογές.

2. Αυτό που πραγματικά ελέγχουν τα εργαλεία αυτά είναι *ασυμφωνίες/αναντιστοιχίες* μεταξύ των ιδιοτήτων που μία υλοποίηση "αναμένεται" να ικανοποιεί με την ίδια την υλοποίηση. Στην πραγματικότητα, οι ιδιότητες αποτελούν το μοντέλο του συστήματος υπό έλεγχο και η πραγματική υλοποίηση ελέγχεται για συμμόρφωση με αυτό το μοντέλο. Στο παράδειγμά μας, ελέγξαμε αν το αποτέλεσμα της αφαίρεσης στοιχείου ικανοποιεί μία μετασυνθήκη αυτού του (υπονοούμενου) μοντέλου.

Οι δοκιμές βάσει μοντέλου καθίστανται ακόμα πιο προφανείς στον έλεγχο βάσει ιδιοτήτων για συστήματα με κατάσταση (stateful PBT), τον οποίο παρουσιάζουμε στη συνέχεια.

2.1 Έλεγχος Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων

Πολλά εργαλεία ελέγχου βάσει ιδιοτήτων παρέχουν τη δυνατότητα μοντελοποίησης συστημάτων με κατάσταση (state machines), η οποία επιτρέπει τον έλεγχο κώδικα με παρενέργειες και συστημάτων που διατηρούν μία εσωτερική κατάσταση κατά τη λειτουργία τους. Σε τέτοιες ρυθμίσεις με κατάσταση, κάθε λειτουργία, που συχνά ονομάζεται εντολή, έχει ένα υπονοούμενο όρισμα που απεικονίζει την κατάσταση, και ένα αόρατο αποτέλεσμα που απεικονίζει τη νέα κατάσταση, κάνοντας τις ιδιότητες δυσκολότερες και ταυτόχρονα λιγότερο ενδιαφέρουσες να διατυπωθούν σε επίπεδο μεμονωμένων λειτουργιών. Αντ'αυτού, τα test cases είναι ακολουθίες εντολών προς το σύστημα υπό έλεγχο. Για αποτελεσματικότητα, η δημιουργία αυτών των ακολουθιών εντολών δε μπορεί να είναι εντελώς τυχαία, αλλά κάθε εντολή πρέπει να έχει νόημα για την κατάσταση στην οποία βρίσκεται το σύστημα υπό έλεγχο· δηλαδή, κάθε εντολή πρέπει να ικανοποιεί μία προσυνθήκη (precondition) για να εμφανιστεί σε ένα συγκεκριμένο σημείο της αλληλουχίας εντολών.

Το PropEr βασίζεται στον έλεγχο βάσει μοντέλου (Model Based Testing) ώστε να ελέγξει συστήματα με κατάσταση [1, 2]. Αυτό σημαίνει ότι το σύστημα ελέγχεται με βάση κάποια μοντελοποίησή του. Παρακάτω μπορούμε να δούμε ότι το PropEr διαιρεί τον έλεγχο συστημάτων με κατάσταση σε δύο φάσεις:

1. Φάση Παραγωγής
2. Φάση Εκτέλεσης

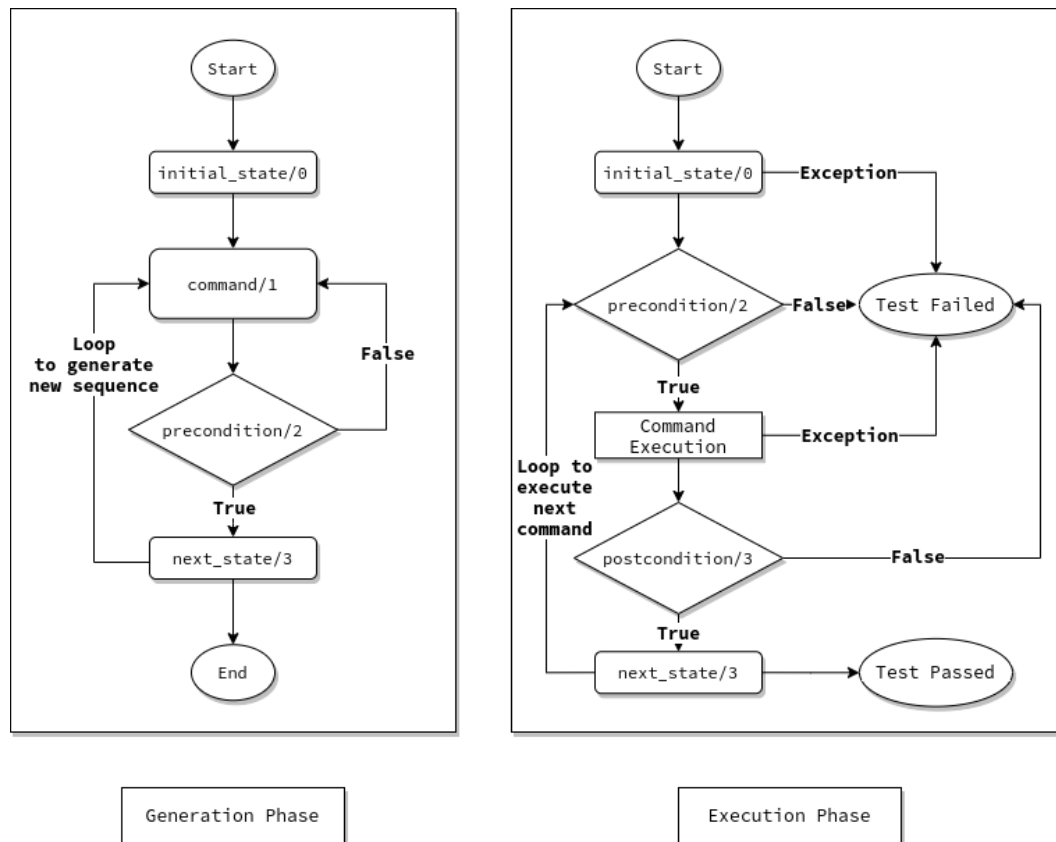
Όπως φαίνεται και στο σχήμα 2.1, στη φάση παραγωγής, η κατάσταση που αποθηκεύεται, οι εντολές και τα ορίσματά τους είναι όλα σε συμβολικό επίπεδο και βοηθούν στην προσομοίωση της λειτουργίας του μοντέλου, καθώς και στην παραγωγή των ακολουθιών εντολών. Στη φάση εκτέλεσης, το PropEr ελέγχει τη συμπεριφορά του πραγματικού συστήματος, η οποία πρέπει να είναι σωστή και σύμφωνη με το μοντέλο του συστήματος.

Όλα τα παραπάνω μπορούν να γίνουν πιο κατανοητά με τη βοήθεια ενός παραδείγματος, εν μέρει εμπνευσμένο από ένα αντίστοιχο σε ένα βιβλίο σχετικά με το PropEr [4, Chapter 9].

2.2 Ένα Κατατοπιστικό Παράδειγμα

Σκεφτείτε μια απλή κρυφή μνήμη (cache) ζευγών κλειδιού-τιμής, η οποία υποστηρίζει τρεις λειτουργίες:

1. Εισαγωγή
2. Αναζήτηση
3. Καθάρισμα



Σχήμα 2.1: Οι φάσεις του Ελέγχου Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων

Μία εισαγωγή στοιχείου με κλειδί το οποίο ήδη υπάρχει στην προσωρινή μνήμη, αντικαθιστά την τιμή του υπάρχοντος στοιχείου στην ίδια θέση. Η προσωρινή μνήμη μπορεί να φιλοξενήσει μόνο ένα μέγιστο αριθμό στοιχείων. Όταν είναι πλήρης, η εισαγωγή ενός στοιχείου με κλειδί το οποίο δεν υπάρχει στη μνήμη, αντικαθιστά το παλαιότερο αντικείμενό της και τοποθετεί το καινούριο αντικείμενο στην τελευταία θέση. Μια αναζήτηση επιστρέφει την τιμή που σχετίζεται με κάποιο κλειδί. Τέλος, το καθάρισμα αφαιρεί όλα τα στοιχεία από την προσωρινή μνήμη.

Η υλοποίηση μιας τέτοιας προσωρινής μνήμης (η οποία δεν παρουσιάζεται εδώ) μπορεί να περιέχει μια πληθώρα ελαττωμάτων και, φυσικά, θέλουμε να τα εντοπίσουμε. Για να εφαρμόσουμε έλεγχο βάσει ιδιοτήτων για αυτό το παράδειγμα, πρώτα πρέπει να καθορίσουμε το *μοντέλο* του συστήματος υπό έλεγχο. Αρχικά, ορίζουμε μία δομή δεδομένων η οποία αντιπροσωπεύει την κατάσταση (`state`) του συστήματος, η οποία περιέχει πληροφορίες για:

1. Το μέγιστο αριθμό στοιχείων που η προσωρινή μνήμη μπορεί να φιλοξενήσει.
2. Τη λίστα των στοιχείων που βρίσκονται αυτή τη στιγμή στη μνήμη.

Η παραπάνω κατάσταση μπορεί να εκφραστεί με την βοήθεια των `records` της `Erlang` σύμφωνα με το πρόγραμμα 2.2.

Πρέπει, τώρα, να ορίσουμε την αρχική κατάσταση του συστήματος ώστε να μπορεί να χρησιμοποιηθεί από το PropEr κατά τη διάρκεια του ελέγχου. Αυτό μπορεί να γίνει με τη συνάρτηση επανάκλησης `initial_state`, όπως ορίζεται στο πρόγραμμα 2.3. Όπως φαίνεται σε αυτό, ορίζουμε ότι στην αρχική κατάσταση η μνήμη μπορεί να έχει έως `?MAX` αντικείμενα και είναι άδεια.

```erlang
-record(state, {max :: integer(), entries :: list()})
```

Πρόγραμμα 2.2: Απεικόνιση Κατάστασης

```erlang
initial_state() ->
  #state{max = ?MAX, entries = []}.
```

Πρόγραμμα 2.3: Συνάρτηση επανάκλησης `initial_state`

Επίσης, πρέπει να ορίσουμε μία λίστα από εντολές που το SUT δέχεται. Στο PropEr, αυτό μπορεί να γίνει με τη συνάρτηση επανάκλησης `command` στο πρόγραμμα 2.4.

```erlang
command(_State) ->
  oneof([{call, ?SUT, insert, [key(), val()]},
         {call, ?SUT, lookup, [key()]},
         {call, ?SUT, flush, []}]).
```

Πρόγραμμα 2.4: Συνάρτηση επανάκλησης `command`

Η συνάρτηση του προγράμματος 2.4 καθορίζει τις συμβολικές κλήσεις των τριών εντολών μέσω των οποίων μπορούμε να αλληλεπιδράσουμε με το SUT, και τα ορίσματα που αυτές οι εντολές δέχονται. Πρέπει να σημειωθεί ότι δε χρησιμοποιούμε την παράμετρο `State` πουθενά στη συνάρτηση, αν και θα μπορούσαμε. Για τη δημιουργία τιμών για αυτά τα ορίσματα, πρέπει να ορίσουμε γεννήτριες για κλειδιά και τιμές. Η γεννήτρια για τιμές δεν είναι ενδιαφέρουσα για το παράδειγμά μας, οποιαδήποτε μας εξυπηρετεί, οπότε θα χρησιμοποιήσουμε ακεραίους για απλότητα. Για τα κλειδιά, θα μπορούσαμε επίσης να χρησιμοποιήσουμε ακεραίους, αλλά μιας και θέλουμε να ελέγξουμε εισαγωγές και αναζητήσεις και με κλειδιά που εμφανίζονται στην προσωρινή μνήμη και με κλειδιά τα οποία δεν εμφανίζονται, θα χρησιμοποιήσουμε τη γεννήτρια του προγράμματος 2.5.

```erlang
key() -> oneof([integer(1, ?MAX), integer()]).
```

Πρόγραμμα 2.5: Γεννήτρια Κλειδιών

Αυτό που κάνει αυτή η γεννήτρια είναι να καθορίσει ότι ένα κλειδί στην περιοχή `[1, MAX]` επιλέγεται με 50% πιθανότητα, ενώ με το υπόλοιπο 50% διαλέγεται κάποιος τυχαίος ακέραιος. Εναλλακτικά, θα μπορούσαμε να είχαμε κάνει τη γεννήτρια κλειδιών παραμετρική ως προς το `State` (δηλαδή `key(State)`) και να διαλέγουμε με 50% πιθανότητα ένα από τα κλειδιά που περιέχει ήδη η προσωρινή μνήμη.

Πρέπει επίσης να μοντελοποιήσουμε τον τρόπο με τον οποίο αλλάζει η κατάσταση μετά την εκτέλεση κάθε εντολής. Αυτό μπορεί να εκφραστεί μέσω της συνάρτησης επανάκλησης `next_state` όπως φαίνεται στο πρόγραμμα 2.6. Ο τρόπος με τον οποίο λειτουργεί η μοντελοποιημένη προσωρινή μνήμη είναι απλός. Αν το αντικείμενο υπάρχει ήδη στη μνήμη, τότε ενημερώνεται η τιμή του. Αν το αντικείμενο δεν υπάρχει στη μνήμη, τότε προστίθεται ως το τελευταίο στοιχείο και αν χρειάζεται (δηλαδή η μνήμη βρίσκεται στο όριο) αφαιρείται το πρώτο.

```
next_state(St, _, {call, ?SUT, insert, [K, V]}) ->
  #state{max = M, entries = L} = St,                   % state before command
  NewL = case lists:keyfind(K, 1, L) of                % construct new entries
           {K,_} -> lists:keyreplace(K, 1, L, {K,V});  % item existed
           false when length(L) < M   -> L ++ [{K,V}]; % add at the end
           false when length(L) =:= M -> tl(L) ++ [{K,V}] % remove head
         end,
  St#state{entries = NewL};                            % return new state
next_state(St, _, {call, ?SUT, lookup, [_K]}) -> St;   % state is unchanged
next_state(St, _, {call, ?SUT, flush, _}) -> St#state{entries = []}.
```

Πρόγραμμα 2.6: Συνάρτηση Επανάκλησης next_state

Ακόμα και μόνο με τα παραπάνω μπορούμε να ελέγξουμε το σύστημα για τυχόν σφάλματα, π.χ. crashes. Για έλεγχο βάσει ιδιοτήτων σε συστήματα με κατάσταση, η ιδιότητα που γράφεται είναι αρκετά τυπική και φαίνεται στο πρόγραμμα 2.7.

```
prop_random_stateful() ->
  ?FORALL(Cmds, commands(?MODULE),
          begin
            ?SUT:start(?MAX),
            {History, State, Result} = run_commands(?MODULE, Cmds),
            ?SUT:stop(),
            ?WHENFAIL(io:format("History: ~p\nState: ~p\nResult: ~p\n",
                                [History, State, Result]),
                      aggregate(command_names(Cmds), Result == ok))
          end).
```

Πρόγραμμα 2.7: Ιδιότητα Προσωρινής Μνήμης

Η κλήση της συνάρτησης commands (γραμμή 2) παράγει τυχαίες ακολουθίες εντολών χρησιμοποιώντας τη συνάρτηση επανάκλησης command, την οποία ορίσαμε παραπάνω. Έπειτα, εκτελούμε αυτές τις εντολές (γραμμή 5) μεταξύ κλήσεων έναρξης και τερματισμού του συστήματος. Αν το Result της εκτέλεσης είναι ok (γραμμή 9), τότε ο έλεγχος είναι επιτυχής και ζητάμε από το PropEr να τυπώσει κάποια στατιστικά σχετικά με τις εντολές (αυτό γίνεται με τη χρήση της συνάρτησης aggregate στη γραμμή 9)· σε περίπτωση αποτυχίας, τυπώνουμε το ιστορικό των εντολών, την κατάσταση του συστήματος και το αποτέλεσμα (γραμμή 7).

Εάν τρέξουμε την ιδιότητα σε αυτό το σημείο, θα δούμε ότι οι τρεις εντολές επιλέγονται με ίση πιθανότητα (33%). Ωστόσο, μερικές από τις ακολουθίες εντολών που παράγονται μπορεί να μην είναι τόσο ενδιαφέρουσες. Στο παράδειγμά μας, δεν έχει νόημα να παράγουμε ακολουθίες εντολών στις οποίες η πρώτη εντολή είναι ο καθαρισμός της μνήμης. Μπορούμε να επηρεάσουμε την ακολουθία εντολών καθορίζοντας κάποιες προσυνθήκες για τις εντολές. Για παράδειγμα (πρόγραμμα 2.8), μπορούμε να καθορίσουμε ότι δεν πρέπει ποτέ να εκτελέσουμε μία εντολή καθαρισμού μνήμης όταν η προσωρινή μνήμη είναι ήδη άδεια.

Μπορούμε, φυσικά, να καθορίσουμε μια παρόμοια προσυνθήκη για την εντολή αναζήτησης ή ακόμα πιο περίπλοκες προσυνθήκες από την παραπάνω. Αν εκτελέσουμε ξανά την ιδιότητα με την παραπάνω μόνο, θα παρατηρήσουμε ότι η εντολή καθαρισμού εμφανίζεται στις ακολουθίες με πιθανότητα $\approx 18\%$, ενώ οι άλλες δύο εντολές με πιθανότητα $\approx 41\%$ η καθεμία.

```
1 precondition(#state{entries = []}, {call, ?SUT, flush, []}) -> false;
2 precondition(_St, {call, _SUT, _Cmd, _Args}) -> true. % no precondition
```

Πρόγραμμα 2.8: Προσυνθήκες

Τέλος, μπορούμε να καθορίσουμε μετασυνθήκες (postconditions) για συγκεκριμένες εντολές, το οποίο θα μας επιτρέψει να ελέγξουμε εάν το σύστημα υπό έλεγχο συμφωνεί με το μοντέλο μας ή όχι. Οι μετασυνθήκες μας επιτρέπουν να προσδιορίσουμε ότι, σε μία κατάσταση πριν την εκτέλεση κάποιας εντολής C, εάν καλέσουμε τη C και πάρουμε κάποιο αποτέλεσμα Res από το σύστημα υπό έλεγχο, αυτό το αποτέλεσμα είναι αυτό που περιμένουμε. Στο πρόγραμμα 2.9, το κάνουμε για τις αναζητήσεις, που είναι οι μόνες εντολές που επιστρέφουν κάτι ενδιαφέρον στο απλό μας παράδειγμα.

```
1 postcondition(#state{entries = L}, {call, ?SUT, lookup, [Key]}, Res) ->
2   case lists:keyfind(Key, 1, L) of
3     {Key, Val} -> Res =:= {ok, Val};
4     false      -> Res =:= {error, not_found}
5   end;
6 postcondition(_St, {call, _SUT, _Cmd, _Args}, _Res) -> true.
```

Πρόγραμμα 2.9: Μετασυνθήκες

Μετά από όλες αυτές τις αλλαγές, ας εκτελέσουμε ξανά την ιδιότητα για ένα σύστημα προσωρινής μνήμης με μέγιστο αριθμό αντικειμένων τα 10. Όπως βλέπουμε παρακάτω, η ιδιότητα περνά πολλές δοκιμές αυξάνοντας την εμπιστοσύνη μας ότι το σύστημα υπό έλεγχο συμφωνεί με το μοντέλο μας (και έτσι το σύστημα υπό έλεγχο λειτουργεί σωστά). Το PropEr μας δείχνει επίσης συγκεντρωτικά ποσοστά εντολών στις παραγόμενες ακολουθίες.

```
1> proper:quickcheck(cache_props:prop_random_stateful(), 10000).
.......................... 10000 dots ..........................
OK: Passed 10000 test(s).

41.15% {cache_impl,lookup,1}
40.89% {cache_impl,insert,2}
17.96% {cache_impl,flush,0}
```

Δυστυχώς, υπάρχει ένα πρόβλημα: το σύστημά μας έχει ένα σφάλμα! Όπως στο παράδειγμα του βιβλίου, η υλοποίηση της προσωρινής μνήμης έχει αρχικοποιηθεί λανθασμένα για να κρατάει $[0, \mathrm{MAX} - 1]$ καταχωρήσεις αντί για $[0, \mathrm{MAX}]$. Πιο κρίσιμα, αυτό το σφάλμα μπορεί να εντοπιστεί στο μοντέλο μας μόνο όταν η προσωρινή μνήμη είναι πλήρης και υπάρχει αναζήτηση ενός αντικειμένου με κλειδί που είναι το ίδιο κλειδί με το στοιχείο που εκδιώχθηκε πιο πρόσφατα από την (υλοποίηση της) προσωρινή μνήμη. Κατά κάποιο τρόπο, ψάχνουμε για μια βελόνα στα άχυρα. Αντί να στηριζόμαστε στην τύχη, παρακάτω θα δούμε πώς ο στοχευμένος έλεγχος βάσει ιδιοτήτων (Targeted Property Based Testing) μπορεί να μας βοηθήσει να το βρούμε.

Κεφάλαιο 3

Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων

Ο στοχευμένος έλεγχος βάσει ιδιοτήτων (targeted property-based testing, ή TPBT) [5], είναι μια παραλλαγή του PBT που στοχεύει στη μείωση του αριθμού των απαιτούμενων δοκιμών ώστε να βρεθεί μια κάποια παραβίαση της ιδιότητας (ή επίτευξη της ίδιας εμπιστοσύνης στο σύστημα υπό έλεγχο) σε σύγκριση με τον τυχαίο έλεγχο βάσει ιδιοτήτων. Αυτό το *επιτυγχάνει χρησιμοποιώντας τεχνικές αναζήτησης ώστε να καθοδηγήσει το σύστημα υπό έλεγχο προς κάποιο στόχο.* Τεχνικά, ο στοχευμένος έλεγχος βάσει ιδιοτήτων αντικαθιστά την τυχαία παραγωγή με μία αναζήτηση με γνώμονα τα αποτελέσματα για εισόδους που έχουν καλύτερες πιθανότητες να βοηθήσουν το σύστημα να χτυπήσει τον στόχο.

Για τον έλεγχο ιδιοτήτων χωρίς κατάσταση, το μόνο που χρειάζεται το TPBT είναι ο χρήστης να ορίσει κάποια *τιμή χρησιμότητας*, δηλαδή μία ποσότητα που υπολογίζεται για το σύστημα και θα μπορέσει να καθοδηγήσει την αναζήτηση, τυπικά μεγιστοποιώντας ή ελαχιστοποιώντας αυτή την τιμή. Πέρα από αυτό, η τεχνική είναι εντελώς αυτόματη [7]. Μία τέτοια ιδιότητα μπορεί να γραφεί ουσιαστικά όπως στο πρόγραμμα 3.1.

```
1  prop_targeted() ->
2    ?FORALL_TARGETED(Input, Generator,       % Use a search strategy.
3                     begin
4                       UV = ?SUT:run(Input), % Run the SUT and find the UV.
5                       ?MAXIMIZE(UV)         % MAXIMIZE the Utility Value.
6                       UV < Threshold        % Test the property.
7                     end).
```

Πρόγραμμα 3.1: Στοχευμένη Ιδιότητα

Η ανάγκη για αυτή την τιμή χρησιμότητας, καθιστά συστήματα τα οποία ελέγχουν μετρικές χρόνου, κατανάλωσης πόρων, ή οποιαδήποτε άλλη ιδιότητα επίδοσης, ιδανικά για στοχευμένο έλεγχο βάσει ιδιοτήτων. Έχοντας ήδη ελέγχους για τέτοιες ιδιότητες, λαμβάνοντας υπόψιν τον κώδικα παραπάνω, είναι πολύ απλό να τους μετατρέψουμε σε στοχευμένους ελέγχους και να αφήσουμε το PropEr να βελτιστοποιήσει την αναζήτηση και να οδηγήσει το σύστημα προς κάποια κατάσταση.

Το κύριο συστατικό του στοχευμένου ελέγχου βάσει ιδιοτήτων, είναι η στρατηγική μεγιστοποίησης (ή ελαχιστοποίησης) της τιμής χρησιμότητας. Το PropEr παρέχει δύο στρατηγικές:

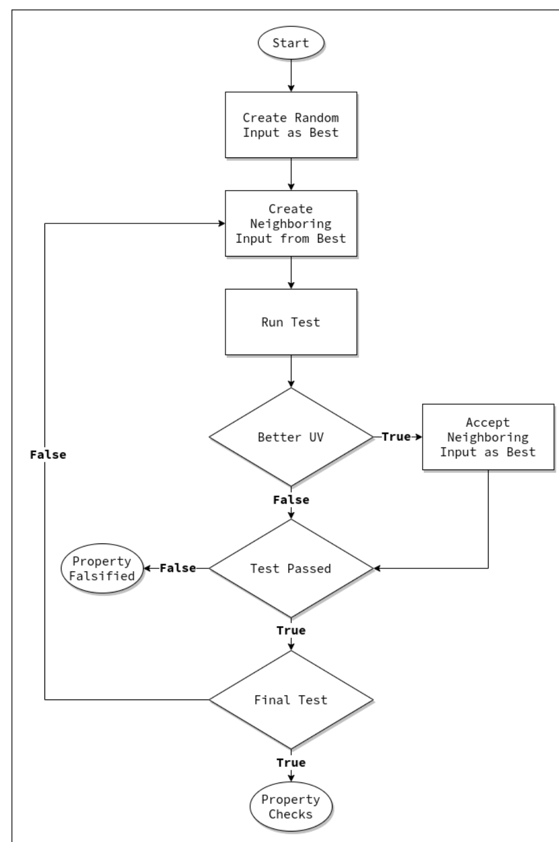
1. Simulated Annealing (default)
2. Hill Climbing

οι οποίες είναι υπεύθυνες να οδηγήσουν την παραγωγή εισόδων προς το στόχο. Ωστόσο, η εσωτερική δομή του κώδικα είναι αρκετά γενική, ώστε να μπορούν να εφαρμοστούν και

άλλες τεχνικές αναζήτησης, για παράδειγμα τεχνικές βασισμένες σε γενετικούς αλγορίθμους ή γραμμικής παλινδρόμησης.

Κάθε στρατηγική που χρησιμοποιείται από το PropEr, χρειάζεται επιπλέον πληροφορίες για να μπορέσει να κατευθύνει τους ελέγχους προς το στόχο που θέτουμε με την τιμή χρησιμότητας. Στην περίπτωση του Simulated Annealing, για παράδειγμα, αυτές οι πληροφορίες συμπεριλαμβάνουν την αρχική συνάρτηση παραγωγής, την τελευταία είσοδο που παράχθηκε καθώς και την καλύτερη είσοδο μέχρι στιγμής. Ωστόσο, η πιο σημαντική πληροφορία είναι η συνάρτηση γειτνίασης (Neighborhood Function ή NF), η οποία είναι υπεύθυνη για την παραγωγή εισόδων που βρίσκονται στη "γειτονιά" της μέχρι τώρα καλύτερης.

Σε κανονικές συνθήκες, το PropEr είναι υπεύθυνο για την παραγωγή των εισόδων. Ωστόσο, όταν χρησιμοποιείται κάποια στρατηγική αναζήτησης, η παραγωγή των εισόδων είναι αρμοδιότητα αυτής, εκμεταλλευόμενη τη συνάρτηση γειτνίασης. Έτσι, ο έλεγχος ξεκινά με μία τυχαία είσοδο από την αρχική συνάρτηση παραγωγής (generator) και συνεχίζει με την παραγωγή γειτονικών εισόδων, οι οποίες γίνονται αποδεκτές ως καλύτερες με γνώμονα την τιμή χρησιμότητας. Το σχήμα 3.1 μας βοηθά να καταλάβουμε τον τρόπο με τον οποίο το PropEr εκμεταλλεύεται τις στρατηγικές αναζήτησης για να επιτύχει τον στοχευμένο έλεγχο βάσει ιδιοτήτων.



Σχήμα 3.1: Στοχευμένος Έλεγχος Συστημάτων με Κατάσταση Βάσει Ιδιοτήτων

Αξίζει να σημειώσουμε, ότι το PropEr δε γνωρίζει τον τρόπο με τον οποίο παράγονται οι γειτονικές είσοδοι, ούτε τον τρόπο με τον οποίο χρησιμοποιείται η τιμή χρησιμότητας. Αυτό συμβαίνει όταν χρησιμοποιούμε την μακροεντολή `?FORALL_TARGETED`, η οποία ενημερώνει το PropEr να μεταφέρει τον έλεγχο παραγωγής στη στρατηγική αναζήτησης, καλώντας τη συνάρτηση η οποία είναι υπεύθυνη να επιστρέψει τον targeted generator,

και τον οποίο θα χρησιμοποιήσει με τον ίδιο τρόπο που θα χρησιμοποιούσε οποιοδήποτε άλλον generator.

3.1 Προσομοίωση Ανόπτησης

Η προσομοίωση ανόπτησης (simulated annealing) [10] είναι μια καλά μελετημένη μεταευριστική τοπική αναζήτηση που χρησιμοποιείται για την αντιμετώπιση διακριτών και, σε μικρότερο βαθμό, προβλημάτων συνεχούς βελτιστοποίησης. Το βασικό χαρακτηριστικό της προσομοιωμένης ανόπτησης είναι ότι παρέχει έναν μηχανισμό για να ξεφύγει από τα τοπικά βέλτιστα επιτρέποντας κινήσεις αναρρίχησης σε λόφο (δηλαδή κινήσεις που επιδεινώνουν την τιμή της αντικειμενικής λειτουργίας) με την ελπίδα να βρεθεί ένα ολικό βέλτιστο.

Η προσομοίωση ανόπτησης λειτουργεί σε ένα χώρο λύσης Ω (το σύνολο των πιθανών λύσεων) και με μία αντικειμενική συνάρτηση $f : \Omega \rightarrow \mathbb{R}$. Επίσης, η προσομοίωση ανόπτησης ελέγχεται από μία επιπλέον παράμετρο, η οποία ονομάζεται θερμοκρασία T και επηρεάζει την πιθανότητα αποδοχής μίας μη βέλτιστης λύσης. Η θερμοκρασία ελαττώνεται σταδιακά όσο ο αλγόριθμος τρέχει, ενώ όσο μεγαλύτερη είναι η θερμοκρασία, τόσο πιο πιθανό είναι να γίνουν αποδεκτές μη βέλτιστες λύσεις. Στο PropEr, το Ω ισούται με όλες τις πιθανές εισόδους I. Η αντικειμενική συνάρτηση f αντιστοιχεί στη συνάρτηση ιδιότητας p, αν αγνοήσουμε το αν η ιδιότητα ισχύει ή όχι. Επιπλέον, η προσομοίωση ανόπτησης ορίζει μία συνάρτηση γειτνίασης $\mathcal{NF} : \Omega \rightarrow \Omega$, η οποία παράγει τυχαίες εισόδους στη "γειτονιά" της αρχικής εισόδου.

Η προσομοίωση ανόπτησης ξεκινά με μία τυχαία λύση από το χώρο λύσεων. Ύστερα, παράγει γειτονικές λύσεις, οι οποίες γίνονται αποδεκτές αν η συσχετισμένη τιμή χρησιμότητας είναι καλύτερη από την μέχρι τώρα καλύτερη, ή με μία πιθανότητα η οποία βασίζεται στην τωρινή θερμοκρασία. Όπως αναφέραμε, όσο ψηλότερη η θερμοκρασία, τόσο πιο πιθανό να γίνει αποδεκτή μία μη βέλτιστη λύση και είναι ο λόγος που βοηθά την τεχνική να ξεφεύγει από τοπικά μέγιστα. Αυτό γίνεται προφανές με τον ορισμό της πιθανότητας αποδοχής:

$$\Pr_{\text{accept}}(i_{n+1}, t_{n+1}) = \begin{cases} e^{-\frac{(uv_n - uv_{n+1})}{t_{n+1}}} & \text{, if } uv_n > uv_{n+1} \\ 1 & \text{, otherwise} \end{cases}$$

Όταν χρησιμοποιούμε την προσομοίωση ανόπτησης ως στρατηγική αναζήτησης, πρέπει να παρέχουμε ένα generator για το I καθώς και μία συνάρτηση γειτνίασης $\mathcal{NF}$ για τον χώρο εισόδων που θέλουμε να χρησιμοποιήσουμε. Ας δούμε πώς το PropEr μας δίνει τη δυνατότητα να ορίσουμε συναρτήσεις γειτνίασης με ένα απλό παράδειγμα, εμπνευσμένο από τις οδηγίες χρήσης του TPBT [6].

3.2 Λαβύρινθος

Ας υποθέσουμε ότι υλοποιούμε ένα παιχνίδι στο οποίο ο παίκτης πρέπει να βρει την έξοδο ενός λαβυρίνθου. Ο λαβύρινθος απεικονίζεται ως ένας πίνακας δύο διαστάσεων, ενώ ο παίκτης μπορεί να κάνει τις εξής κινήσεις: 1. μετακίνηση ένα τετράγωνο πάνω, 2. μετακίνηση ένα τετράγωνο δεξιά, 3. μετακίνηση ένα τετράγωνο κάτω, 4. μετακίνηση ένα τετράγωνο αριστερά, ενώ ταυτόχρονα υπάρχουν και τοίχοι σε τυχαία τετράγωνα.

Στη γλώσσα του PropEr η συνάρτηση παραγωγής μονοπατιών μπορεί να γραφτεί με τη βοήθεια των συναρτήσεων του προγράμματος 3.2.

```
1 step() ->
2   oneof([up, right, down, left]).
3
4 path() ->
5   list(step()).
```

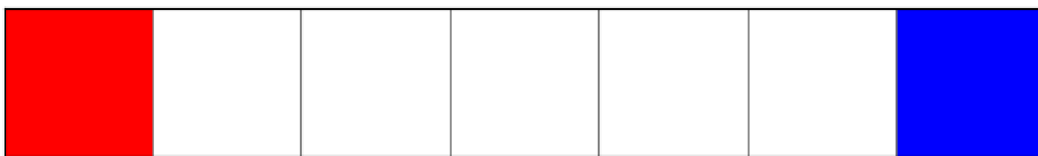
Πρόγραμμα 3.2: Συνάρτηση Παραγωγής Μονοπατιών

Υποθέτουμε, επίσης, ότι έχουμε μια συνάρτηση η οποία ακολουθεί το μονοπάτι, με βάση τον λαβύρινθο, και ενημερώνει αν ο παίκτης βρήκε την έξοδο. Έτσι, η ιδιότητα που θέλουμε να ελέγξουμε, είναι αν υπάρχει μονοπάτι που οδηγεί στην έξοδο και γράφεται όπως στο πρόγραμμα 3.3.

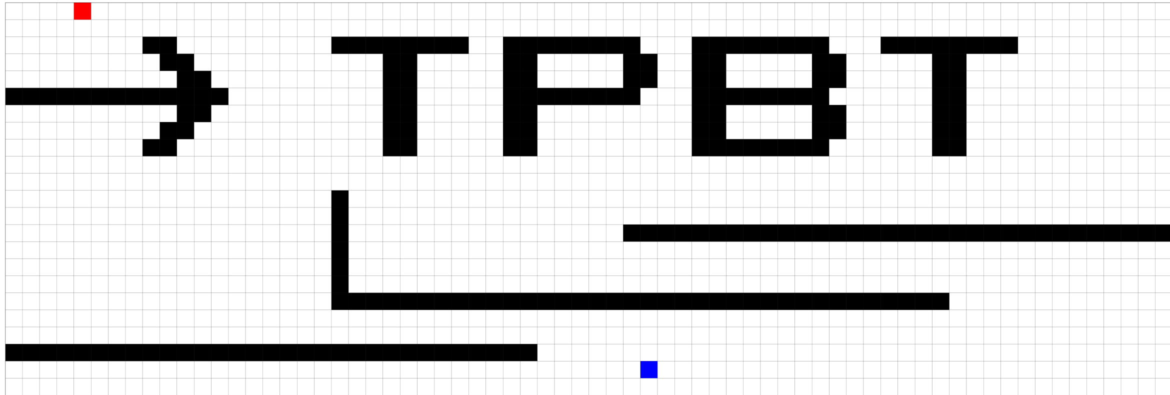
```
1 prop_random(Maze) ->
2   #{entrance := Entrance} = Maze,
3   ?FORALL(Path, path(),
4           case follow_path(Entrance, Path, Maze) of
5             {exited, _} -> false
6             _           -> true
7           end).
```

Πρόγραμμα 3.3: Ιδιότητα Λαβυρίνθου

Θα ελέγξουμε την ιδιότητα αυτή με δύο διαφορετικούς λαβυρίνθους. Στην πρώτη περίπτωση (σχήμα 3.2) ο λαβύρινθος είναι και μικρός και “εύκολος”, ενώ στη δεύτερη περίπτωση (σχήμα 3.3) αποτελείται από περισσότερα εμπόδια, είναι μεγαλύτερος και εν γένει πιο “δύσκολος”.



Σχήμα 3.2: Εύκολος Λαβύρινθος



Σχήμα 3.3: Δύσκολος Λαβύρινθος

Στα σχήματα 3.2 και 3.3, τα άσπρα τετράγωνα αντιπροσωπεύουν θέσεις από τις οποίες μπορεί να περάσει ο παίκτης, τα μαύρα τετράγωνα είναι εμπόδια, ενώ η είσοδος είναι μπλε και η έξοδος κόκκινη. Τρέχοντας την ιδιότητα που ορίσαμε, λαμβάνουμε τα εξής αποτελέσματα.

```
1> proper:quickcheck(labyrinth:prop_random(MazeEasy)).
.........................................!
Failed: After 42 test(s).
[left,up,down,left,left,down,left,left,left,up,left]

Shrinking ...(3 time(s))
[left,left,left,left,left,left]

2> proper:quickcheck(labyrinth:prop_random(MazeHard), [10000]).
.......................... 10000 dots .........................
OK: passed 10000 test(s).
```

Εδώ αξίζει να σημειωθεί ότι οι κινήσεις γίνονται χωρίς κάποια προσυνθήκη και η διαχείρισή τους γίνεται από την συνάρτηση που "ακολουθεί" το μονοπάτι. Για αυτό το λόγο στο πρώτο παράδειγμα παράγονται και βήματα down και up παρόλο που στην πραγματικότητα δεν ακολουθούνται.

Παρατηρούμε ότι το PropEr βρήκε εύκολα λύση για τον "εύκολο" λαβύρινθο σε μόλις 42 δοκιμές. Αντιθέτως, για το "δύσκολο" λαβύρινθο, ακόμα και μετά από 10000 δοκιμές, το PropEr δεν κατάφερε να βρει κάποιο μονοπάτι που να οδηγεί στην έξοδο. Το πρόβλημα αυτό, ωστόσο, μπορεί να λυθεί πολύ πιο εύκολα αν χρησιμοποιήσουμε στοχευμένο έλεγχο βάσει ιδιοτήτων.

Όπως αναφέραμε στην ενότητα 3.1, για να χρησιμοποιήσουμε στοχευμένο έλεγχο, χρειαζόμαστε μία τιμή χρησιμότητας καθώς και μία συνάρτηση γειτνίασης. Η τιμή χρησιμότητας στην περίπτωσή μας, είναι η απόσταση της τελικής θέσης του μονοπατιού από την έξοδο, την οποία και θέλουμε να μηδενίσουμε. Έπειτα, η συνάρτηση γειτνίασης μπορεί να γραφεί στο PropEr όπως στο πρόγραμμα 3.4.

```
1 path_nf(Base, {_Depth, _Temperature}) ->
2   ?LET(NextPath, vector(20, step()), Base ++ NextPath).
```

Πρόγραμμα 3.4: Συνάρτηση Γειτνίασης Μονοπατιών

Αυτή η συνάρτηση γειτνίασης ορίζει πως, δεδομένου ενός `Base` και μίας πλειάδας που περιέχει το βάθος (`Depth`) και τη θερμοκρασία (`Temperature`), θα επιστρέψει μια συνάρτηση παραγωγής (generator), ή μια τιμή σε άλλες περιπτώσεις. Το όρισμα `Base` αντιστοιχεί στη μέχρι τώρα *βέλτιστη* είσοδο για τη στρατηγική αναζήτησης. Το `Depth` είναι ένα όρισμα που χρησιμοποιείται εσωτερικά από το PropEr για να καταλάβει το βάθος στο οποίο βρίσκεται στις διάφορες εμφωλευμένες συναρτήσεις παραγωγής. Τέλος, το `Temperature` είναι η θερμοκρασία της προσομοίωσης, με εύρος τιμών στο $[0, 1]$, με την αρχική τιμή να είναι το 1 με σταδιακή μείωση προς το 0. Στην περίπτωσή μας, επιστρέφεται μια νέα συνάρτηση παραγωγής (generator), η οποία παράγει μονοπάτια τα οποία προκύπτουν αν προσθέσουμε στο ήδη καλύτερο μονοπάτι άλλα 20 τυχαία βήματα. Έτσι, τα μονοπάτια συνεχώς μεγαλώνουν και ταυτόχρονα κινούνται πιο κοντά στην έξοδο.

Το PropEr παρέχει αυτόματα συναρτήσεις γειτνίασης (με μεγαλύτερη πολυπλοκότητα από αυτή του προγράμματος 3.4) για όλους τους σημαντικούς τύπους, αλλά δίνει τη δυνατότητα και στους χρήστες να ορίσουν τη δική τους συνάρτηση γειτνίασης. Αυτό μπορεί να γίνει μέσω της μακροεντολής **`?USERNF`**, η οποία παίρνει δύο ορίσματα:

1. Τη συνάρτηση παραγωγής της αρχικής τιμής.
2. Τη συνάρτηση γειτνίασης.

Με βάση όλα όσα αναφέραμε, η ιδιότητα του παραδείγματός μας τώρα μετατρέπεται στο πρόγραμμα 3.5.

```
1  prop_targeted(Maze) ->
2    #{entrance := Entrance, exit := Exit} = Maze,
3    ?FORALL_TARGETED(Path, ?USERNF(path(), fun path_nf/2),
4                     case follow_path(Entrance, Path, Maze) of
5                       {exited, _} -> false;
6                       Pos ->
7                         UV = distance(Pos, Exit),
8                         ?MINIMIZE(UV),
9                         true
10                    end).
```

Πρόγραμμα 3.5: Στοχευμένη Ιδιότητα Λαβυρίνθου

Τρέχοντας ξανά την ιδιότητα, με τη χρήση του στοχευμένου ελέγχου βάσει ιδιοτήτων, παρατηρούμε ότι πλέον το PropEr είναι ικανό να βρει λύση στο πρόβλημα, σε μόνο 57 προσπάθειες (`FORALL_TARGETED` και `MAXIMIZE` είναι μακροεντολές που παρέχονται από το PropEr).

```
1> proper:quickcheck(labyrinth:prop_targeted(MazeHard)).
........................................................!
Failed: After 57 test(s).
[up,right,right,up,up,...,up,left]

Shrinking .............. 70 dots ..............(70 time(s))
[left,left,up,left,left,up,left,left,left,left,left,left,left,
 left,left,left,up,left,left,left,left,left,up,up,up,up,up,up,
 up,up,up,up,left,up,up,left,up,left,up,left,left,up,left,up,
 left,left,left,left,left,left,left,up,left]
```

Το παραπάνω πρόβλημα μας βοηθά να κατανοήσουμε την ευκολία που παρέχει το PropEr στο χρήστη ώστε να παρέχει μία δική του συνάρτηση γειτνίασης ή και σε γενικότερες γραμμές να χρησιμοποιεί το στοχευμένο έλεγχο βάσει ιδιοτήτων, με πολύ λίγες αλλαγές στον κώδικα των ιδιοτήτων.

Κεφάλαιο 4

Στοχευμένος Έλεγχος Βάσει Ιδιοτήτων σε Συστήματα με Κατάσταση

Επιστρέφοντας στον έλεγχο συστημάτων με κατάσταση, θα θέλαμε φυσικά και έναν τρόπο να επεκτείνουμε τον στοχευμένο έλεγχο βάσει ιδιοτήτων και σε τέτοια συστήματα. Σε αυτό το κεφάλαιο, εισάγουμε νέες συναρτήσεις, οι οποίες βοηθούν στην παραγωγή ακολουθιών εντολών με τη βοήθεια του στοχευμένου ελέγχου. Με αυτές, μπορούμε να μετατρέψουμε το παράδειγμά μας (ενότητα 2.2), ώστε να χρησιμοποιεί τον στοχευμένο έλεγχο του PropEr (πρόγραμμα 4.1).

```
1  prop_targeted_stateful() ->
2    ?FORALL_TARGETED(
3       Cmds, commands(?MODULE),
4       begin
5         ?SUT:start(?MAX),
6         {History, State, Result} = run_commands(?MODULE, Cmds),
7         ?SUT:stop(),
8         ?MAXIMIZE(length(State#state.entries)),
9         ?WHENFAIL(io:format("History: ~p\nState: ~p\nResult: ~p\n",
10                            [History, State, Result]),
11                  aggregate(command_names(Cmds), Result == ok))
12      end).
```

Πρόγραμμα 4.1: Στοχευμένος Έλεγχος Προσωρινής Μνήμης

Συγκρίνοντας με την ιδιότητα για τυχαίο έλεγχο βάσει ιδιοτήτων (πρόγραμμα 2.7), η κύρια αλλαγή εδώ είναι η προσθήκη στη γραμμή 8, στην οποία ορίζεται ότι ο έλεγχος θα πρέπει να κατευθύνεται προς την μεγιστοποίηση καταστάσεων στις οποίες υπάρχουν πολλές εγγραφές στην προσωρινή μνήμη. Η μόνη άλλη, αν και συνήθης αλλαγή, είναι η αλλαγή στη γραμμή 2, όπου ορίζεται ότι η παραγωγή των ακολουθιών των εντολών είναι στοχευμένη. Αν τώρα τρέξουμε τη στοχευμένη ιδιότητα έχουμε το εξής:

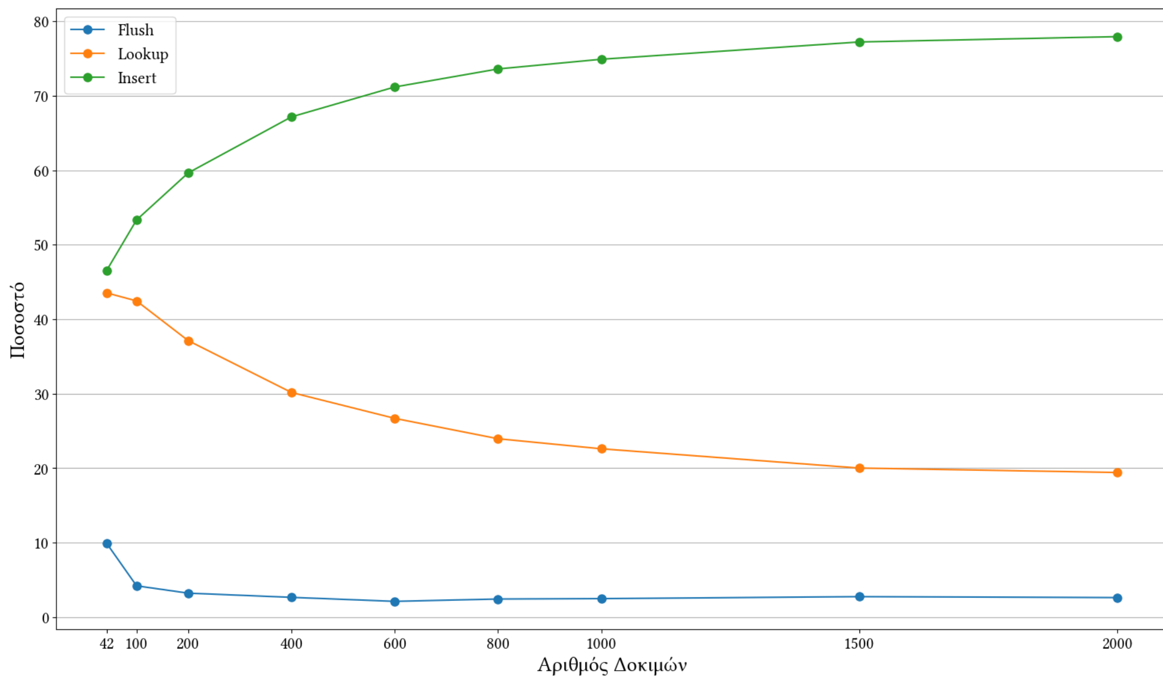
```
1> proper:quickcheck(cache_props:prop_targeted_stateful()).
...........................................................................!
Failed: After 76 test(s).
[{init,{state,10,0,[]}},{set,{var,1},{call,cache_statem,cache,[6,-4]}},...,
 {error,not_found}}]
State: {state,10,10,[{4,1},{1,0},{5,6},{8,-62},{10,-11},{-5,-46},{29,11},
                     {-18,-4},{16,-7},{-23,67}]}
Result: {postcondition,false}

Shrinking ..............(14 time(s))
[{init,{state,10,0,[]}},
```

```
  {set,{var,3},{call,cache_statem,cache,[4,1]}},
  {set,{var,4},{call,cache_statem,cache,[1,0]}},
  {set,{var,12},{call,cache_statem,cache,[-5,-46]}},
  {set,{var,14},{call,cache_statem,cache,[29,11]}},
  {set,{var,16},{call,cache_statem,cache,[5,-16]}},
  {set,{var,17},{call,cache_statem,cache,[10,-11]}},
  {set,{var,18},{call,cache_statem,cache,[-18,-4]}},
  {set,{var,21},{call,cache_statem,cache,[16,-7]}},
  {set,{var,23},{call,cache_statem,cache,[-23,67]}},
  {set,{var,29},{call,cache_statem,cache,[8,-62]}},
  {set,{var,30},{call,cache_statem,find,[4]}}]
History: [{{state,10,0,[]},true},
          {{state,10,1,[{4,1}]},true},
          {{state,10,2,[{4,1},{1,0}]},true},
          {{state,10,3,[{4,1},{1,0},{-5,-46}]},true},
          {{state,10,4,[{4,1},{1,0},{-5,-46},{29,11}]},true},
          {{state,10,5,[{4,1},{1,0},{-5,-46},{29,11},{5,-16}]},true},
          {{state,10,6,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11}]},true},
          {{state,10,7,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11},
                        {-18,-4}]},true},
          {{state,10,8,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11},
                        {-18,-4},{16,-7}]},true},
          {{state,10,9,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11},
                        {-18,-4},{16,-7},{-23,67}]},true},
          {{state,10,10,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11},
                         {-18,-4},{16,-7},{-23,67},{8,-62}]},
                         {error,not_found}}]
State: {state,10,10,[{4,1},{1,0},{-5,-46},{29,11},{5,-16},{10,-11},{-18,-4},
                     {16,-7},{-23,67},{8,-62}]}
Result: {postcondition,false}
false
```

Κάνοντας την παραγωγή ακολουθιών εντολών στοχευμένη, φέρει πολύ καλό αποτέλεσμα στο απλό μας παράδειγμα: το σφάλμα ανιχνεύεται με πιθανότητα 42% όταν τρέχουμε μόλις 100 δοκιμές, και αρκετά συχνά με λιγότερες. Θα παρουσιάσουμε τους ακριβείς μηχανισμούς που χρησιμοποιούνται στις ακόλουθες ενότητες, αλλά η ουσία είναι ότι το PropEr χρησιμοποιεί την προσομοίωση ανόπτησης (την προεπιλεγμένη στρατηγική του στοχευμένου ελέγχου) για να δώσει σταδιακά θετική μεροληψία σε εντολές που οδηγούν σε καταστάσεις συστήματος, όπου ο αριθμός των στοιχείων της προσωρινής μνήμης αυξάνεται, και σε αρνητική προκατάληψη έναντι των εντολών που τον μειώνουν.

Τέλος, είναι σημαντικό να παρατηρήσουμε ότι η μόνη ενέργεια που έκανε ο χρήστης ήταν να καθορίσει την *τιμή χρησιμότητας*, δηλαδή μία τιμή που παράγεται από μία συνάρτηση ορισμένων στοιχείων της κατάστασης, η οποία ενθυλακώνει την κατεύθυνση στην οποία πρέπει να στοχεύει η υποκείμενη στρατηγική. Το πώς αυτή η κατεύθυνση αντανακλάται στις ακολουθίες των εντολών είναι εντελώς αδιαφανής στο εργαλείο δοκιμών. Όπως ο τυχαίος, έτσι και ο στοχευμένος έλεγχος βάσει ιδιοτήτων είναι μία τεχνική *μαύρου κουτιού (black box)*: δεν έχει καμία γνώση για τη σημασιολογία των εντολών στο σύστημα υπό έλεγχο, ούτε αναλύει τον κώδικα με οποιονδήποτε τρόπο. Στο παράδειγμά μας, το PropEr μαθαίνει αυτόματα ότι οι εντολές εισαγωγής αυξάνουν την τιμή χρησιμότητας, οι αναζητήσεις την αφήνουν αμετάβλητη και οι καθαρισμοί τη μειώνουν. Αυτή η ιδέα επεκτείνεται φυσικά και σε πιο περίπλοκα συστήματα με κατάσταση και σύνολα εντολών.

Στο σχήμα 4.1, παρουσιάζουμε αντιπροσωπευτικά ποσοστά εντολών, όπως ευρέθησαν από τη συσσωμάτωση που καλείται στην ιδιότητα, κατά την εκτέλεση, καθώς και την πιθανότητα εύρεσης του σφάλματος, για διάφορες τιμές δοκιμών.



Σχήμα 4.1: Ποσοστό Εμφάνισης Εντολών

Συγκρίνοντάς τα, τόσο με αυτά της ενότητας 2.2 όσο και μεταξύ τους, παρατηρούμε ότι η παραγωγή μετατοπίζεται σταδιακά στην προτίμηση ακολουθιών εντολών με όλο και περισσότερες εισαγωγές και όλο και λιγότερους καθαρισμούς. Επιπροσθέτως, αυτή η διαδικασία σταματάει όταν η κρυφή μνήμη γεμίσει – στο οποίο σημείο θα βρεθεί και το σφάλμα σύντομα. Τέλος, παρατηρούμε ότι όσο αυξάνονται οι δοκιμές, τα ποσοστά εμφάνισης της εντολής εισαγωγής συνεχίζει να αυξάνεται, ενώ το ποσοστό των ευρέσεων και καθαρισμού μειώνεται. Στο κεφάλαιο 5 θα αναφέρουμε με περισσότερες λεπτομέρειες τα αποτελέσματα της αξιολόγησης του προβλήματος της προσωρινής μνήμης που χρησιμοποιήθηκε στο πλαίσιο της παρούσας εργασίας.

4.1 Αλγόριθμος

Για να μπορέσει να δουλέψει ο στοχευμένος έλεγχος σε συστήματα με κατάσταση, πρέπει να παρέχουμε μια συνάρτηση γειτνίασης στην στρατηγική αναζήτησης. Για να δημιουργήσουμε μία τέτοια συνάρτηση, πρέπει να σκεφτούμε έναν αλγόριθμο, ο οποίος, δεδομένης μίας `Base` ακολουθίας εντολών και της θερμοκρασίας της προσομοίωσης ανόπτησης, πρέπει να παράγει μια άλλη ακολουθία εντολών στη γειτονιά της βάσης.

Ο αλγόριθμος είναι, εν μέρει, εμπνευσμένος από τον αλγόριθμο dd_{min} [12] και βασίζεται σε δύο βασικές λειτουργίες πάνω σε ακολουθίες. Αυτό σημαίνει, ότι, δεδομένης μίας βασικής ακολουθίας εντολών, η οποία είναι και η *καλύτερη έως τώρα*, μπορούμε να εκτελέσουμε δύο διαφορετικές λειτουργίες:

1. Επέκταση της ακολουθίας
2. Μετάλλαξη της ακολουθίας

Ενώ η πρώτη επιλογή είναι αρκετά απλή, καθώς επιτυγχάνεται με την προσθήκη επιπλέον εντολών στην ακολουθία, η δεύτερη είναι πιο περίπλοκη λόγω της φύσης και του

τρόπου παραγωγής των ακολουθιών των εντολών. Όπως αναφέραμε και προηγουμένως (ενότητα 2.1), οι ακολουθίες εντολών παράγονται σύμφωνα με κάποιες *προσυνθήκες*, οι οποίες εξαρτώνται από την κατάσταση του συστήματος σε κάθε σημείο της ακολουθίας. Αν μεταλλάξουμε τυχαία αυτή την ακολουθία, τότε υπάρχει περίπτωση να ακυρώσουμε όλες τις εντολές που ακολουθούν τη μεταλλαγμένη. Για να προσπεράσουμε αυτό το μειονέκτημα, η μετάλλαξη γίνεται μόνο στην ουρά της ακολουθίας των εντολών, αφαιρώντας έναν τυχαίο αριθμό εντολών από αυτή και έπειτα ακολουθώντας την πρώτη λειτουργία, δηλαδή την προσθήκη επιπλέον εντολών από το σημείο αυτό και μετά. Παρακάτω θα δούμε πώς αυτό μεταφράζεται με τη βοήθεια των λειτουργιών που παρέχει το PropEr, καθώς και τον τρόπο με τον οποίο επιτυγχάνεται η διαδικασία της συρρίκνωσης (shrinking).

4.2 Υλοποίηση

Η υλοποίηση του αλγορίθμου, με τον PropEr τρόπο, είναι λίγο πιο δύσκολη, επειδή, όπως αναφέραμε, θέλουμε να μπορούμε να επιτυγχάνουμε συρρίκνωση των εισόδων μας. Αρχικά, ας δούμε πώς παράγονται οι τυχαίες ακολουθίες εντολών με τη βοήθεια του προγράμματος 4.2.

```
1  commands_gen(Mod) ->
2    ?LET(InitialState, ?LAZY(Mod:initial_state()),
3         ?SUCHTHAT(
4            Cmds,
5            ?LET(List,
6                 ?SIZED(Size,
7                        proper_types:noshrink(
8                          commands_gen(Size * ?RESIZE_FACTOR,
9                                       Mod, InitialState, 1))),
10                proper_types:shrink_list(List)),
11           is_valid(Mod, InitialState, Cmds, []))).
```

Πρόγραμμα 4.2: Συνάρτηση Παραγωγής `commands_gen/1`

Αυτή η συνάρτηση παραγωγής (generator) όριζει πως αφού παραχθεί η αρχική κατάσταση (γραμμή 2), θα παράγει ακολουθίες εντολών, χρησιμοποιώντας τη βοηθητική συνάρτηση παραγωγής `commands_gen/4` (γραμμή 8), οι οποίες δε θα είναι μεγαλύτερες σε μήκος από την παράμετρο `Size`. Μετά από αυτή την παραγωγή, οι ακολουθίες θα συρρικνωθούν με βάση τη γραμμή 10, η οποία δίνει εντολή στο PropEr να προσπαθήσει απλά να αφαιρέσει στοιχεία από την ακολουθία, ενώ ταυτόχρονα τηρείται η προϋπόθεση ότι είναι μια έγκυρη ακολουθία εντολών για το σύστημα υπό έλεγχο (γραμμή 11).

Η κύρια ιδέα της συνάρτησης παραγωγής `commands_gen/4` (πρόγραμμα 4.3), είναι ότι θα παράγει εντολές αναδρομικά και θα τις προσθέτει στην ακολουθία. Πιο συγκεκριμένα, είτε θα παράγει μία νέα εντολή (γραμμή 5), σύμφωνα με τις *προϋποθέσεις* (γραμμή 6) της τωρινής κατάστασης, και μετά θα εκτελέσει τον εαυτό της, ή θα επιστρέψει μία άδεια λίστα (base case της αναδρομής). Η πιθανότητα παραγωγής μιας νέας εντολής εξαρτάται από την τιμή της παραμέτρου μεγέθους (`Size`), καθώς στη γραμμή 3 δίνουμε εντολή στο PropEr να επιλέξει μεταξύ των δοσμένων συναρτήσεων παραγωγής με βάρος. Επίσης, αυτή η συνάρτηση παραγωγής, σε συνδυασμό με τη μακροεντολή `LAZY` (γραμμή 2), είναι που βοηθάνε το PropEr να συρρικνώσει την ακολουθία των εντολών και είναι *σημαντικό* να σημειώσουμε, και για την υλοποίηση για το στοχευμένο έλεγχο, ότι η απλή αναδρομή *δε* θα παρήγαγε ακολουθίες με δυνατότητα συρρίκνωσης.

```
commands_gen(Size, Mod, State, Count) ->
  ?LAZY(
     proper_types:frequency(
       [{1, []},
        {Size, ?LET(Call,
                    ?SUCHTHAT(X, Mod:command(State),
                              Mod:precondition(State, X)),
                    begin
                      Var = {var, Count},
                      NextState = Mod:next_state(State, Var, Call),
                      ?LET(
                         Cmds,
                         commands_gen(Size - 1, Mod, NextState, Count + 1),
                         [{set, Var, Call} | Cmds])
                    end)}])).
```

Πρόγραμμα 4.3: Συνάρτηση Παραγωγής commands_gen/4

Όπως είδαμε παραπάνω, στην ενότητα 3.1, για να μπορέσει το PropEr να εκτελέσει στοχευμένο έλεγχο χρειάζεται να του παρέχουμε, με τη βοήθεια της μακροεντολής USERNF, την αρχική συνάρτηση παραγωγής καθώς και τη συνάρτηση γειτνίασης. Εισάγουμε στο πρόγραμμα 4.4 τη συνάρτηση παραγωγής commands/1, η οποία χρησιμοποιήθηκε στο παράδειγμα και καθοδηγεί το PropEr να χρησιμοποιήσει στοχευμένο έλεγχο[1]:

```
commands(Mod) ->
  ?USERNF(commands_gen(Mod), next_commands_gen(Mod)).
```

Πρόγραμμα 4.4: Συνάρτηση Παραγωγής commands/1

Παρατηρούμε ότι η αρχική συνάρτηση παραγωγής είναι η ήδη υπάρχουσα συνάρτηση παραγωγής τυχαίων ακολουθιών, ενώ η συνάρτηση γειτνίασης είναι μία νέα συνάρτηση. Με οδηγό τις παραπάνω υλοποιήσεις, παρουσιάζουμε την υλοποίηση του αλγορίθμου που εισήχθη σε αυτή τη διπλωματική, ως μια συνάρτηση παραγωγής του PropEr, στο πρόγραμμα 4.5. Αναλύοντας τη συνάρτηση αυτή, μπορούμε ξεκάθαρα να παρατηρήσουμε ότι επιστρέφει μία συνάρτηση γειτνίασης, η οποία ακολουθεί τα ίδια βήματα με την απλή συνάρτηση παραγωγής. Η κύρια διαφορά βρίσκεται στη γραμμή 10 και η χρήση μίας επιπλέον παραμέτρου (γραμμή 3). Η παράμετρος αυτή “αποφασίζει” τον μέγιστο αριθμό εντολών που θα αφαιρεθούν από την ουρά της βασικής ακολουθίας.

[1] Σε προηγούμενες εκδόσεις του PropEr, η συνάρτηση commands/1 ταυτιζόταν με τη συνάρτηση commands_gen/1 που παρουσιάσαμε στο πρόγραμμα 4.2

```
next_commands_gen(Mod) ->
  fun (Cmds, {_Depth, Temp}) ->
      MaxRemovals = round(length(Cmds) * Temp),
      ?LET(InitialState, ?LAZY(Mod:initial_state()),
           ?SUCHTHAT(
              NewCmds,
              ?LET(List,
                   ?SIZED(Size,
                          proper_types:noshrink(
                            next_commands_gen(Mod, InitialState, Cmds,
                                              Size * ?RESIZE_FACTOR,
                                              MaxRemovals))),
                   proper_types:shrink_list(List)),
              is_valid(Mod, InitialState, NewCmds, [])))
  end.
```

Πρόγραμμα 4.5: Συνάρτηση Παραγωγής `next_commands_gen/1`

Στο πρόγραμμα 4.6, παρουσιάζουμε και τις βοηθητικές συναρτήσεις που χρησιμοποιήθηκαν για την υλοποίηση στο πρόγραμμα 4.5. Η βασική ιδέα αυτών των δύο συναρτήσεων παραγωγής είναι ότι λειτουργούν αρμονικά ώστε να παράγουν μία καινούρια ακολουθία εντολών, η οποία βασίζεται στην προηγουμένως παραχθείσα. Εάν το μήκος αυτής της ακολουθίας είναι μικρότερο από την παράμετρο μεγέθους (γραμμή 4), τότε η συνάρτηση απλά θα προσθέσει κάποιες επιπλέον εντολές στη βασική ακολουθία. Σε άλλη περίπτωση, θα προσπαθήσει να αφαιρέσει μερικές εντολές από την ουρά της ακολουθίας (γραμμή 9) και μόνο μετά να προσθέσει επιπλέον εντολές. Πρέπει, επίσης, να σημειώσουμε ότι για την προσθήκη επιπλέον εντολών στην ακολουθία, χρησιμοποιείται η απλή συνάρτηση παραγωγής `commands_gen/4` (γραμμή 17). Αυτή η συνάρτηση παραγωγής, παράγει ακολουθίες εντολών που μπορούν να συρρικνωθούν επειδή τόσο η `remove_commands/2` όσο και η `commands_gen/4` είναι "τεμπέλικες" αναδρομικές συναρτήσεις παραγωγής, οι οποίες, όπως αναφέραμε ήδη, είναι το κλειδί στην προσπάθεια του PropEr να επιτύχει συρρίκνωση (shrinking).

Για λόγους πληρότητας, παραθέτουμε όλες τις συναρτήσεις που απαρτίζουν το δημόσιο API του PropEr και τις οποίες αναφέραμε πως εισάγαμε.

```erlang
next_commands_gen(Mod, InitialState, Cmds, Size, MaxRemovals) ->
  %% Try to remove commands after the maximum current size is reached.
  Threshold = case Size > length(Cmds) of
                  true -> 0
                  false -> MaxRemovals
              end,
  ?LET(LessCmds,
       ?LET(LessRCmds,
            remove_cmds(lists:reverse(Cmds), Threshold),
            lists:reverse(LessRCmds)),
       ?LET(CmdsTail,
            begin
              Length = length(LessCmds),
              Remaining = Size - Length,
              State = state_after(Mod, [{init, InitialState} | LessCmds]),
              Count = Length + 1,
              commands_gen(Remaining, Mod, State, Count)
            end,
            LessCmds ++ CmdsTail)).

remove_cmds(Cmds, Threshold) ->
  ?LAZY(proper_types:frequency(
          [{1, proper_types:exactly(Cmds)},
           {Threshold, ?LAZY(remove_cmds(tl(Cmds), Threshold - 1))}])).
```

Πρόγραμμα 4.6: Συνάρτηση Παραγωγής next_commands_gen/5

```erlang
commands(Mod) ->
  ?USERNF(commands_gen(Mod), next_commands_gen(Mod)).

commands(Mod, InitialState) ->
  ?USERNF(commands_gen(Mod, InitialState),
          next_commands_gen(Mod, InitialState)).
```

Πρόγραμμα 4.7: Δημόσιο API proper_statem

```
commands(Mod) ->
  ?USERNF(commands_gen(Mod), next_commands_gen(Mod)).

commands(Mod, InitialState) ->
  ?USERNF(commands_gen(Mod, InitialState),
          next_commands_gen(Mod, InitialState)).

% private
next_commands_gen(Mod) ->
  InitialState = initial_state(Mod),
  StatemNext = proper_statem:next_commands_gen(?MODULE, InitialState),
  fun (Cmds, {Depth, Temp}) ->
      ?LET([_ | NextCmds], StatemNext(Cmds, {Depth, Temp}), NextCmds)
  end.

% private
next_commands_gen(Mod, {Name, Data} = InitialState) ->
  State = #state{name = Name, data = Data, mod = Mod},
  StatemNext = proper_statem:next_commands_gen(?MODULE, State),
  fun (Cmds, {Depth, Temp}) ->
      ?LET([_ | NextCmds], StatemNext(Cmds, {Depth, Temp}),
           [{init, InitialState} | NextCmds])
  end.
```

Πρόγραμμα 4.8: Δημόσιο API proper_fsm

```
next_commands_gen(Mod, InitialState) ->
  fun Aux([{init, _S} | Cmds], {Depth, Temp}) ->
      Aux(Cmds, {Depth, Temp});
      Aux(Cmds, {_Depth, Temp}) ->
      MaxRemovals = round(length(Cmds) * Temp),
      ?SUCHTHAT(
         NewCmds,
         ?LET(CmdsTail,
              ?LET(List,
                   ?SIZED(Size,
                          proper_types:noshrink(
                            next_commands_gen(Mod, InitialState, Cmds,
                                              Size * ?RESIZE_FACTOR,
                                              MaxRemovals))),
                   proper_types:shrink_list(List)),
              [{init, InitialState} | CmdsTail]),
         is_valid(Mod, InitialState, NewCmds, []))
  end.
```

Πρόγραμμα 4.9: Συνάρτηση Παραγωγής next_commands_gen/2

Κεφάλαιο 5

Πειραματική Αξιολόγηση

Σε αυτό το κεφάλαιο, θα παρουσιάσουμε τα αποτελέσματα από διάφορα πειράματα που εκτελέσαμε για το παράδειγμα της προσωρινής μνήμης, σε μορφή διαγραμμάτων. Τα διαγράμματα αυτά, χωρίζονται σε δύο κατηγορίες.

1. Στην πρώτη κατηγορία είναι τα διαγράμματα που παρουσιάζουν με μία καμπύλη το μέσο όρο των μετρικών που παρουσιάζονται.
2. Στη δεύτερη κατηγορία είναι τα διαγράμματα που παρουσιάζουν την κατανομή των μετρικών που παρατηρήθηκε κατά τη διάρκεια της εκτέλεσης των πειραμάτων.

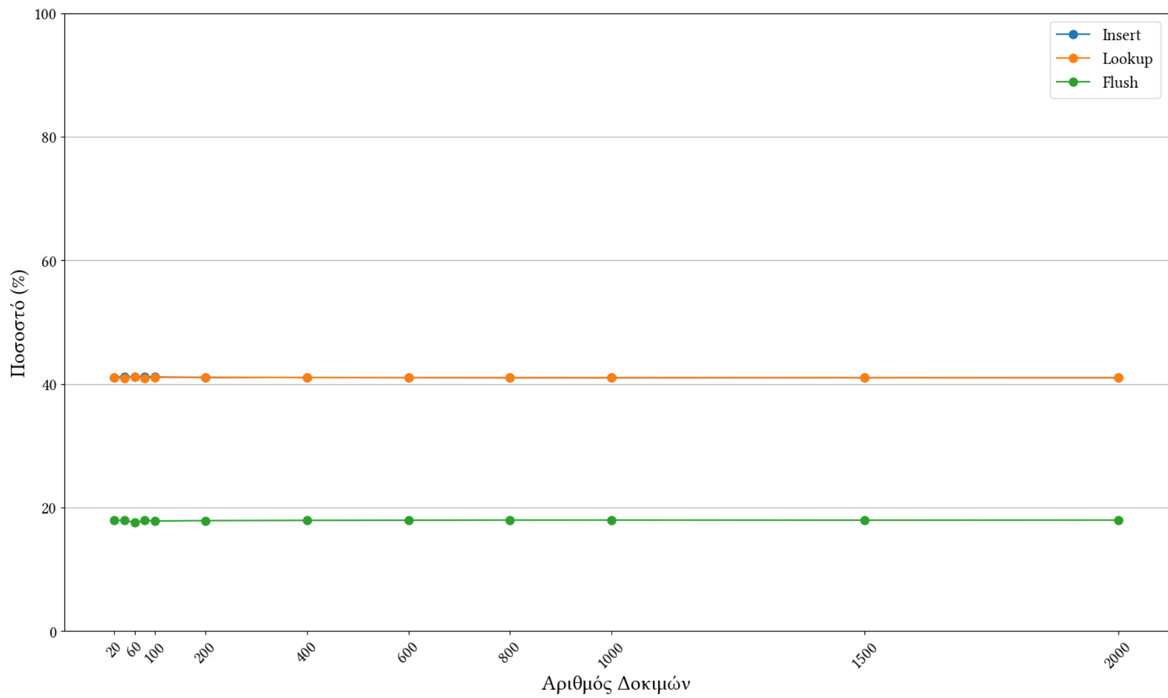
Επίσης, τα πειράματα που εκτελέσαμε χωρίζονται και αυτά σε δύο κατηγορίες.

1. Πειράματα με σκοπό την παρατήρηση της διαμόρφωσης των ποσοστών εντολών αναλόγως της παραμέτρου μέγιστου πλήθος δοκιμών του PropEr.
2. Πειράματα με σκοπό την παρατήρηση της διαμόρφωσης της πιθανότητας εύρεσης του σφάλματος του παραδείγματος αναλόγως τόσο της παραμέτρου μέγιστου πλήθους δοκιμών του PropEr όσο και του μεγέθους της προσωρινής μνήμης.

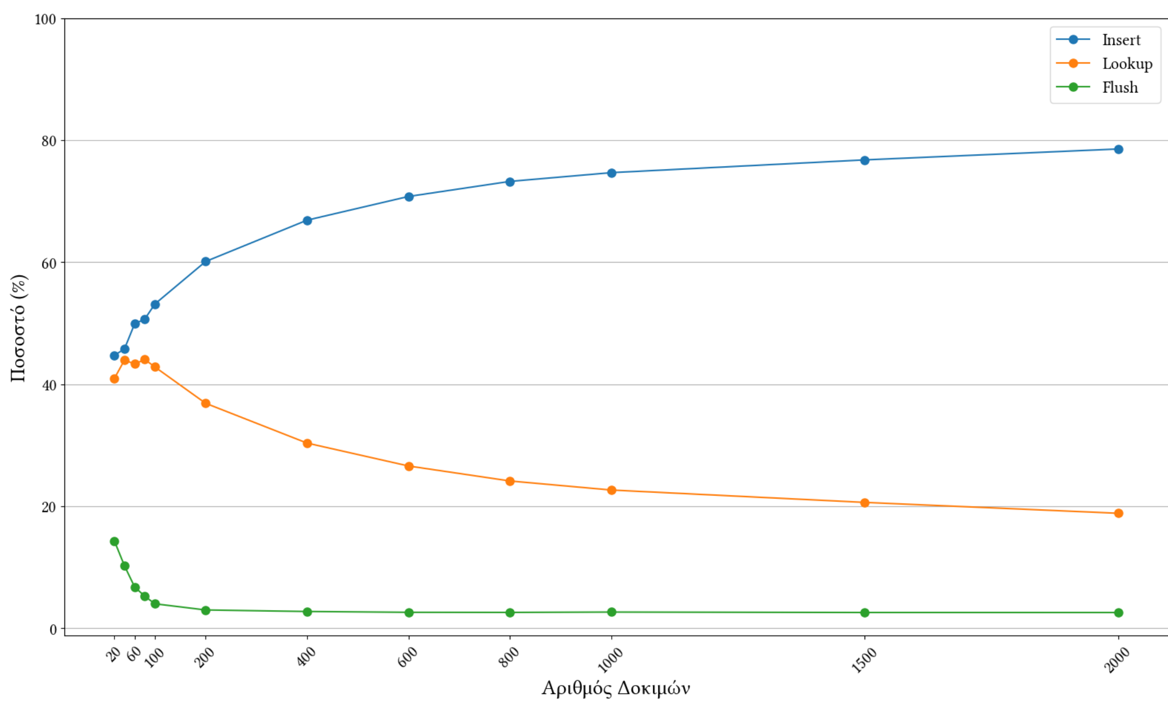
5.1 Συχνότητα Εμφάνισης Εντολών

Τα πρώτα πειράματα που εκτελέσαμε είχαν ως σκοπό τη μέτρηση της συχνότητας εμφάνισης κάθε εντολής καθώς το PropEr εκτελούσε δοκιμές για το σύστημα της προσωρινής μνήμης. Η παράμετρος που εξετάζεται σε αυτά τα πειράματα είναι το πλήθος δοκιμών που εκτελεί το PropEr, ενώ το μέγεθος της μνήμης ορίστηκε σε κάτι αρκετά μεγάλο ώστε να μην επηρεάζει την εμφάνιση των εντολών (όπως στην περίπτωση που η μνήμη είναι ήδη γεμάτη, ο στοχευμένος έλεγχος δε θα διάλεγε επιπλέον προσθήκες). Για κάθε τιμή της παραμέτρου εκτελέσαμε 1000 πειράματα στα οποία μετρήσαμε τον αριθμό εμφάνισης κάθε εντολής και έπειτα μετατρέψαμε σε ποσοστά. Τα πειράματα εκτελέστηκαν τόσο για τον τυχαίο έλεγχο (πρόγραμμα 2.7) όσο και για τον στοχευμένο έλεγχο (πρόγραμμα 4.1).

Αρχικά θα παρουσιάσουμε το μέσο όρο του ποσοστού εμφάνισης κάθε εντολής για κάθε τιμή της παραμέτρου για τα δύο είδη ελέγχου. Στο σχήμα 5.1, παρατηρούμε ότι ο αρχικός μας ισχυρισμός ισχύει. Πράγματι οι εντολές `insert` και `lookup` εμφανίζονται σε ποσοστό $\approx 41\%$, ενώ η εντολή `flush` με ποσοστό $\approx 18\%$. Αντίθετα, στο σχήμα 5.2, παρατηρούμε ότι η εντολή `flush` κυμαίνεται στο $2-4\%$ (για αριθμό δοκιμών άνω των 100), ενώ το ποσοστό εμφάνισης των εντολών `insert` και `lookup` μεταβάλλεται αναλόγως με τον αριθμό των δοκιμών που τρέχει το PropEr. Η εντολή `insert` εμφανίζεται όλο και πιο συχνά όσο περισσότερες δοκιμές γίνονται, ενώ το αντίθετο συμβαίνει για την εντολή `lookup`.



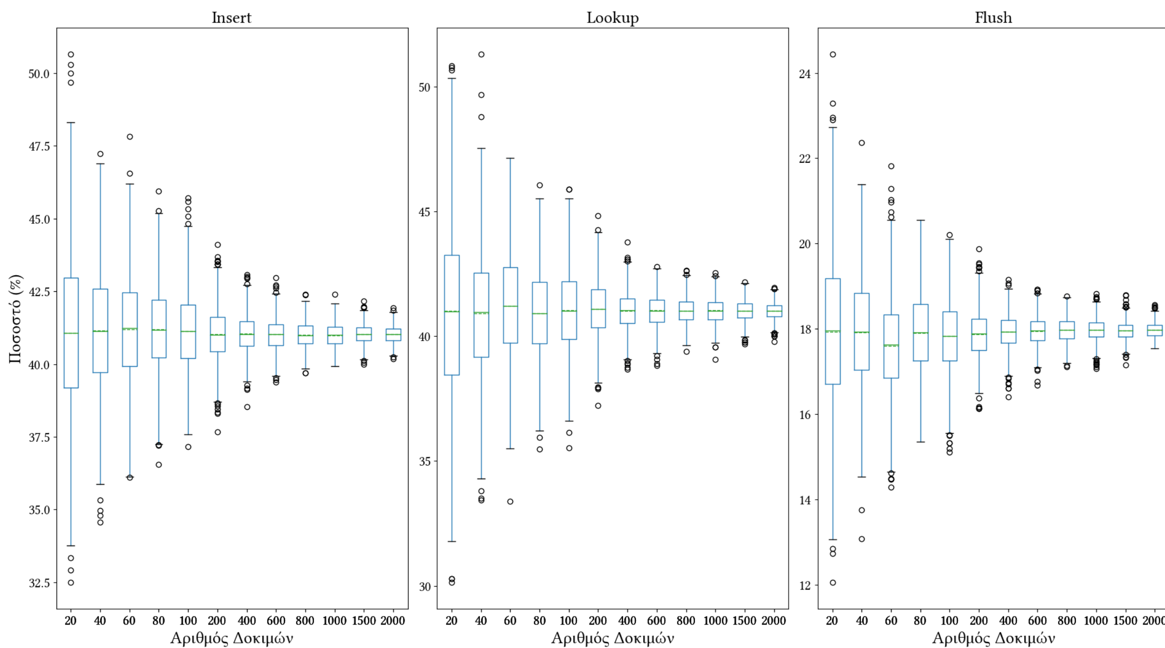
Σχήμα 5.1: Ποσοστό Εμφάνισης Εντολών στον Τυχαίο Έλεγχο



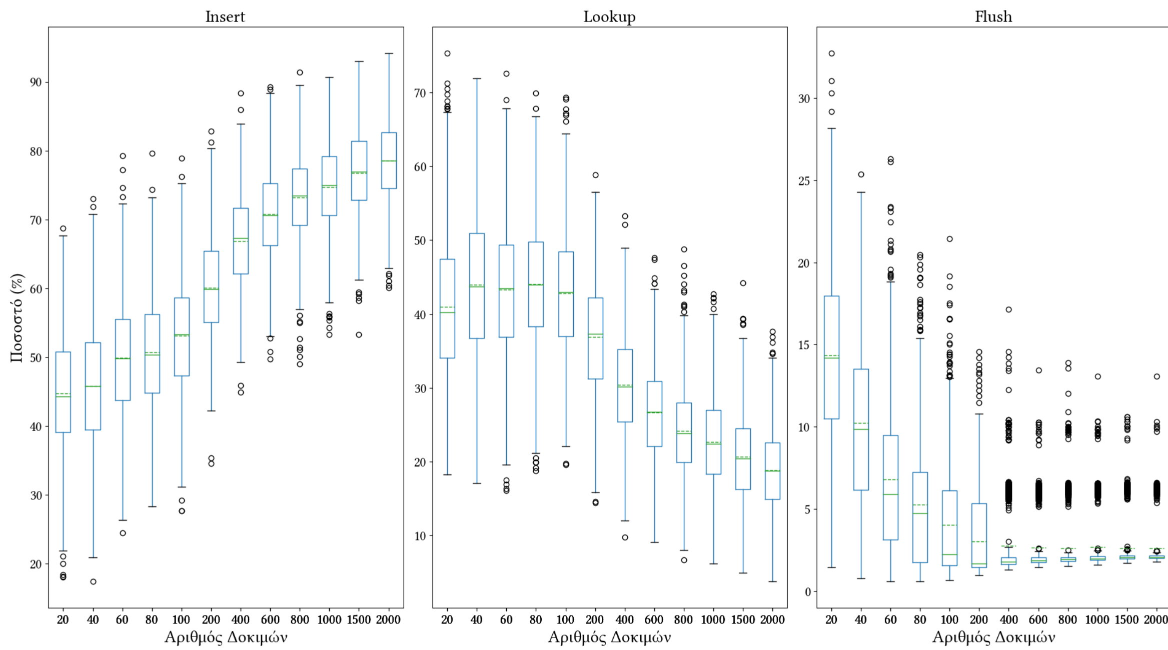
Σχήμα 5.2: Ποσοστό Εμφάνισης Εντολών στον Στοχευμένο Έλεγχο

Η συμπεριφορά που παρατηρείται είναι η ίδια με αυτή που θα προσπαθούσαμε να επιτύχουμε με χειροκίνητο τρόπο αν θέλαμε να γεμίσουμε την προσωρινή μνήμη, αλλά σε αυτή την περίπτωση η διαδικασία είναι ήδη αυτόματη. Επίσης, χάρη στα διαγράμματα αυτά, λαμβάνουμε μία πρώτη εντύπωση για την απόδοση του αλγορίθμου που υλοποιήσαμε, καταλαβαίνοντας ότι όσες περισσότερες δοκιμές εκτελούνται τόσο καλύτερη είναι η αναζήτηση και τόσο πιο κοντά βρίσκεται στην εύρεση του ολικού βέλτιστου.

Στη συνέχεια, με τη βοήθεια των σχημάτων 5.3 και 5.4, παρουσιάζουμε την κατανομή του ποσοστού εμφάνισης κάθε εντολής. Για τον τυχαίο έλεγχο (σχήμα 5.3), τα αποτελέσματα είναι αναμενόμενα. Όσο περισσότερες δοκιμές εκτελούνται τόσο μικρότερη είναι η διακύμανση του ποσοστού εμφάνισης κάθε εντολής, με την τάση να σταθεροποιηθεί πολύ κοντά στον μέσο όρο που παρουσιάστηκε στο σχήμα 5.1. Αξίζει, επίσης, να σημειωθεί ότι οι αξιοσημείωτες τιμές, οι οποίες αποτυπώνονται στο διάγραμμα ως κύκλοι, είναι πολύ λίγες. Αντίθετα, στον στοχευμένο έλεγχο (σχήμα 5.4), παρατηρούμε ότι η διακύμανση του ποσοστού εμφάνισης κάθε εντολής είναι αρκετά υψηλή, ενώ οι αξιοσημείωτες τιμές είναι πολύ περισσότερες. Αυτό συμβαίνει διότι η προσομοίωση ανόπτησης προσπαθεί να αναζητήσει ένα μεγάλο χώρο του προβλήματος και της κατανομής των εντολών, ώστε εν τέλει να βρει το βέλτιστο. Για αυτό το λόγο, πολλές φορές δοκιμάζει λύσεις οι οποίες μπορεί να μην είναι κοντά στο ολικό βέλτιστο, αλλά μπορεί να βοηθήσουν να ξεφύγει από κάποιο τοπικό βέλτιστο. Έτσι, τα αποτελέσματα του σχήματος 5.4 είναι τα αναμενόμενα.



Σχήμα 5.3: Κατανομή Ποσοστού Εμφάνισης Εντολών στον Τυχαίο Έλεγχο

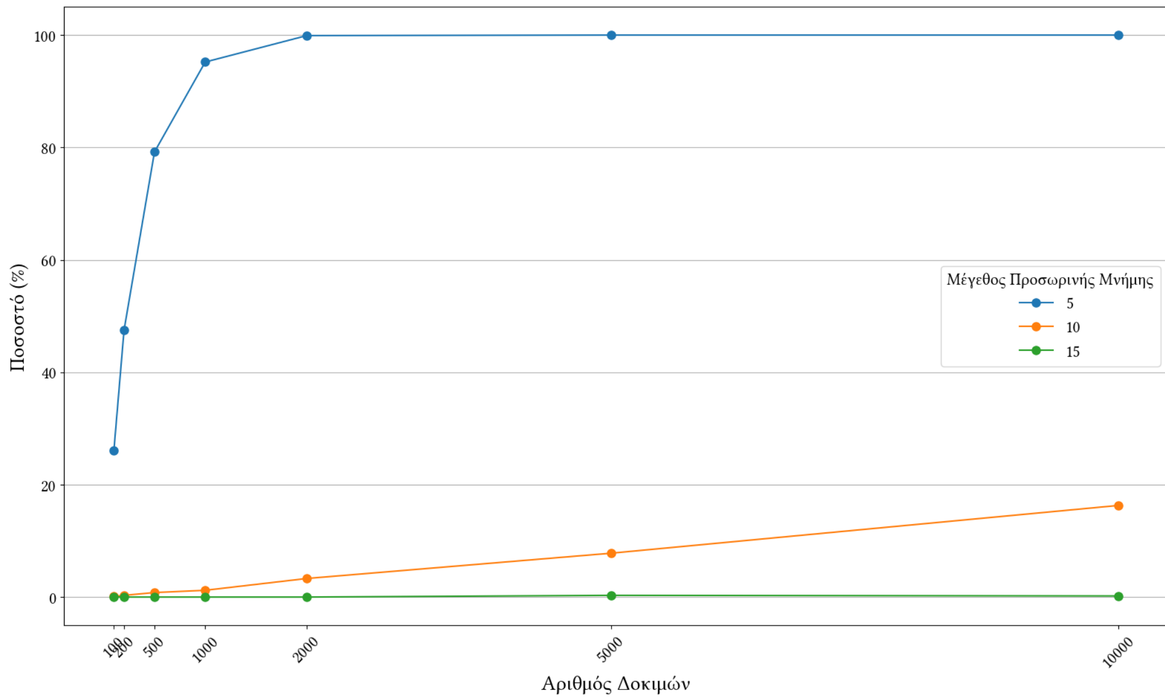


Σχήμα 5.4: Κατανομή Ποσοστού Εμφάνισης Εντολών στον Στοχευμένο Έλεγχο

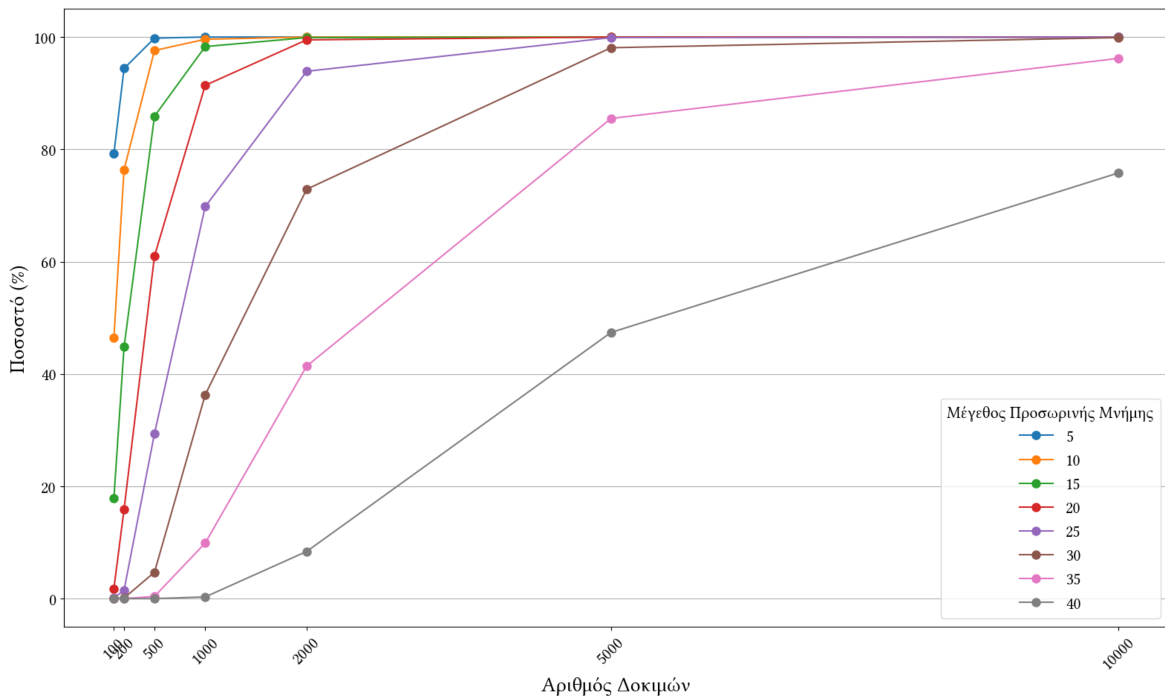
5.2 Πιθανότητα Εύρεσης Σφάλματος

Τα πειράματα αυτής της ενότητας εκτελέστηκαν με σκοπό την καταγραφή της εύρεσης του σφάλματος στο παράδειγμα της προσωρινής μνήμης. Σε αυτή την περίπτωση, οι παράμετροι που εξετάζονται είναι πλέον δύο, 1. το πλήθος των δοκιμών που θα τρέξει το PropEr, 2. το μέγεθος της προσωρινής μνήμης, με σκοπό την εξέταση του τρόπου με τον οποίο αυτές επηρεάζουν την πιθανότητα εύρεσης του σφάλματος. Για κάθε συνδυασμό των δύο αυτών παραμέτρων, εκτελέστηκαν 1000 πειράματα. Αξίζει, επίσης, να σημειωθεί ότι για όλα τα πειράματα αυτής της ενότητας, το μέγιστο πλήθος εντολών που μπορεί να παράξει το PropEr τέθηκε στις 50 εντολές.

Αρχικά, παρουσιάζουμε με τα σχήματα 5.5 και 5.6, την πιθανότητα εύρεσης του σφάλματος για κάθε μέγεθος προσωρινής μνήμης προς τον αριθμό των δοκιμών του PropEr. Η πιθανότητα προκύπτει ως η διαίρεση του αριθμού των πειραμάτων που οδήγησαν σε εύρεση του σφάλματος προς το συνολικό αριθμό πειραμάτων. Για την περίπτωση του τυχαίου ελέγχου (σχήμα 5.5), τα αποτελέσματα είναι όπως αναφέραμε και στην ενότητα 2.2, δηλαδή για μέγεθος μνήμης τουλάχιστον 10, το PropEr δεν μπορεί να εντοπίσει το σφάλμα με συνέπεια, ενώ ήδη για μεγέθη μνήμης > 15 δεν μπορεί να το εντοπίσει καθόλου. Αντίθετα, στον στοχευμένο έλεγχο (σχήμα 5.6), η πιθανότητα εύρεσης του σφάλματος είναι πολύ καλύτερη, τόσο ώστε το PropEr μπορεί να εντοπίσει το σφάλμα ακόμα και για μέγεθος μνήμης 40. Το κύριο συμπέρασμα που λαμβάνουμε, όμως, από αυτό το διάγραμμα είναι πόσο σημαντικό ρόλο παίζει ο αριθμός των δοκιμών, όταν χρησιμοποιείται στοχευμένος έλεγχος. Παρατηρούμε ότι όσο περισσότερες δοκιμές εκτελεί το PropEr τόσο μεγαλύτερη συνέπεια παρουσιάζει στην εύρεση του σφάλματος. Και αυτή η συμπεριφορά είναι αναμενόμενη, καθώς η προσομοίωση ανόπτησης φέρει πάντα καλύτερα αποτελέσματα ανάλογα με το πλήθος των επαναλήψεων της διαδικασίας, καθώς έτσι εξερευνεί ένα μεγαλύτερο εύρος του χώρου του προβλήματος.

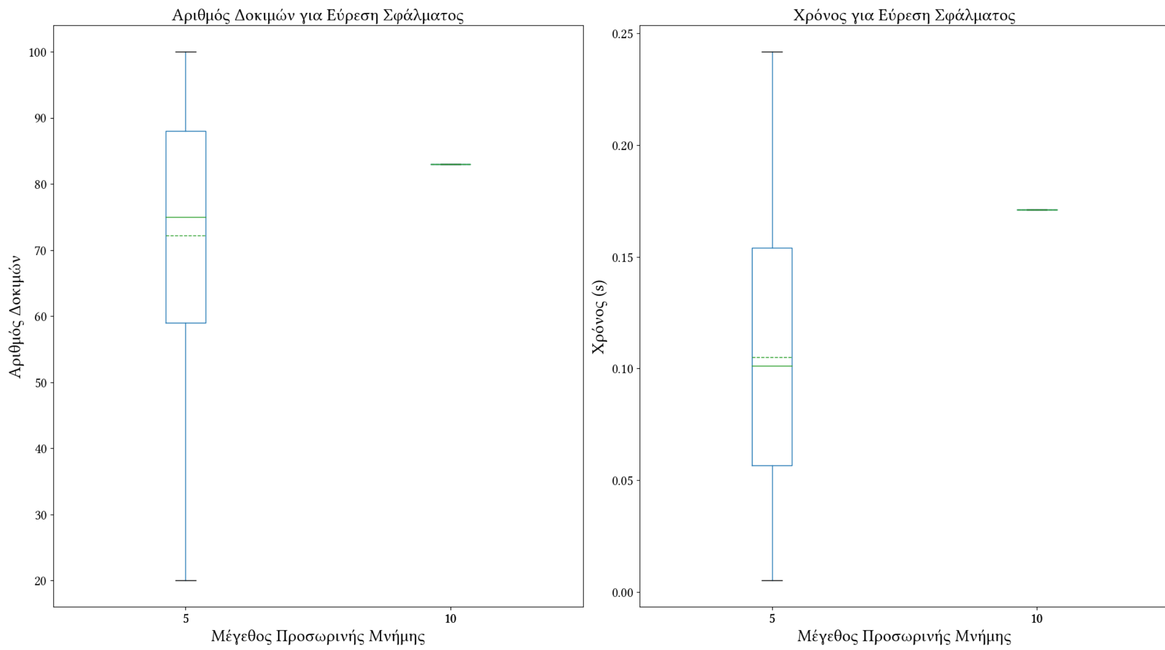


Σχήμα 5.5: Πιθανότητα Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο

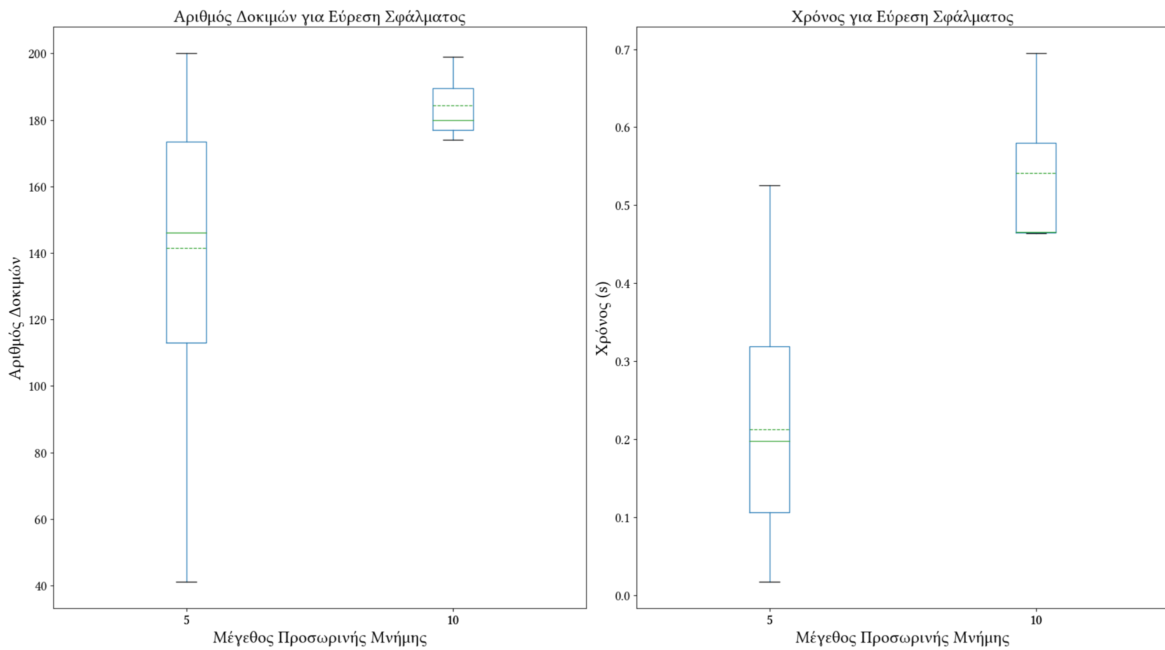


Σχήμα 5.6: Πιθανότητα Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο

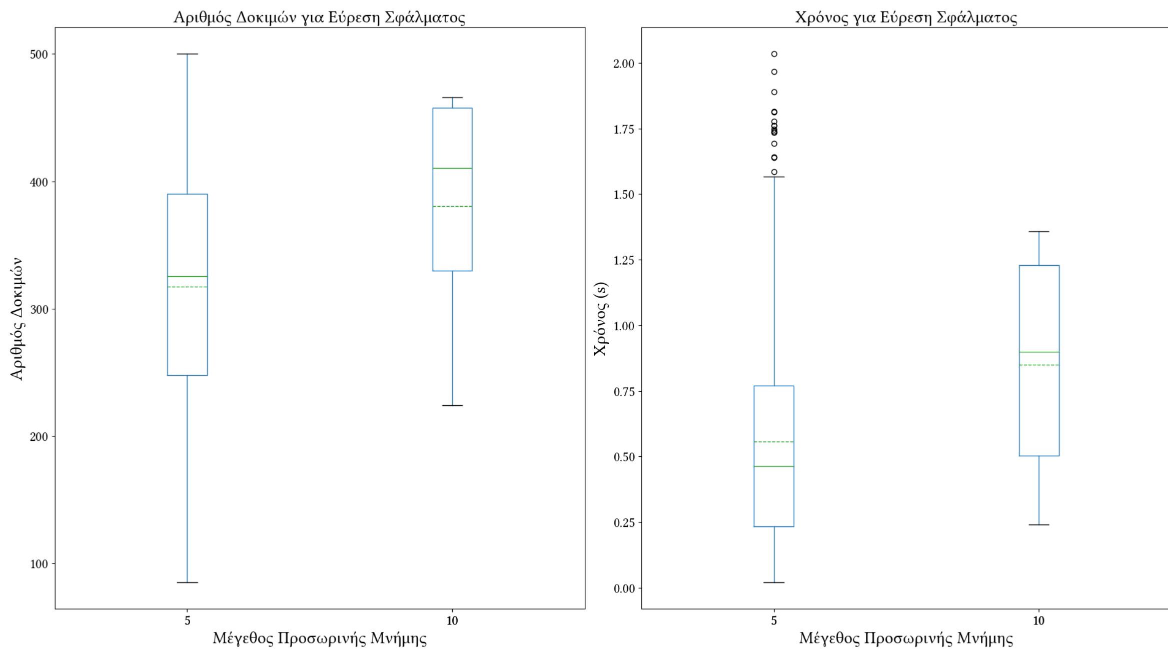
Παρά το γεγονός ότι η χρήση μεγαλύτερου αριθμού δοκιμών φέρει καλύτερα ποσοστά εύρεσης του σφάλματος, υπάρχει το μειονέκτημα του χρόνου. Όσο περισσότερες δοκιμές εκτελεί το PropEr τόσο πιο αργή είναι η διαδικασία, ενώ, ταυτόχρονα, η προσομοίωση ανόπτησης προσθέτει ακόμα περισσότερη πολυπλοκότητα στη διαδικασία κοστίζοντας ακόμα περισσότερο χρόνο. Έτσι, για κάθε πρόβλημα πρέπει να αξιολογείται το πλήθος των δοκιμών που χρειάζεται να εκτελεστούν για να βρεθεί το σφάλμα, αλλά και το πλήθος αυτών που πραγματικά εκτελέστηκαν ώστε να βρεθεί, δηλαδή σε ποιά δοκιμή βρέθηκε το σφάλμα, και ο χρόνος που χρειάζεται για να γίνει αυτό. Παρακάτω, παρουσιάζουμε διαγράμματα που μας βοηθάνε να καταλάβουμε μια τάξη μεγέθους για το χρόνο που χρειάζεται ώστε να βρεθεί το σφάλμα, αλλά και για τον αριθμό των δοκιμών για να συμβεί αυτό. Χωρίς να αναλύσουμε κάθε διάγραμμα ξεχωριστά, μπορούμε να επιβεβαιώσουμε τον ισχυρισμό μας πως όσο περισσότερες δοκιμές εκτελούμε τόσο περισσότερο χρόνο χρειαζόμαστε για να βρούμε το σφάλμα. Το ίδιο συμβαίνει και όσο αυξάνεται το μέγεθος της προσωρινής μνήμης.



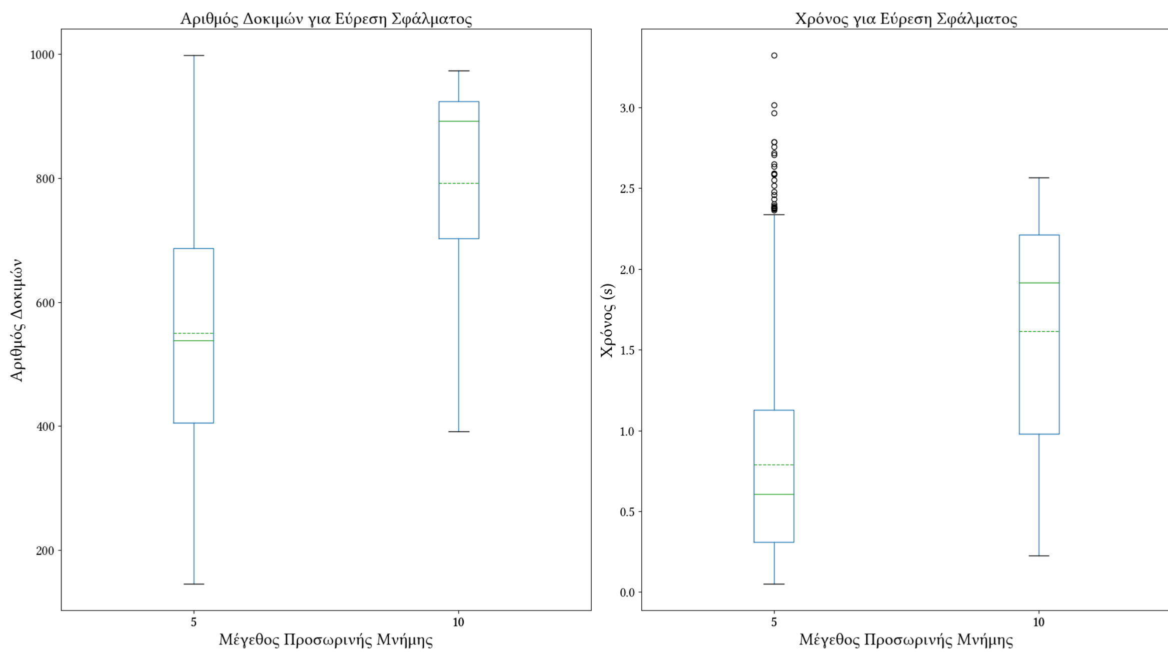
Σχήμα 5.7: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (100 Δοκιμές)



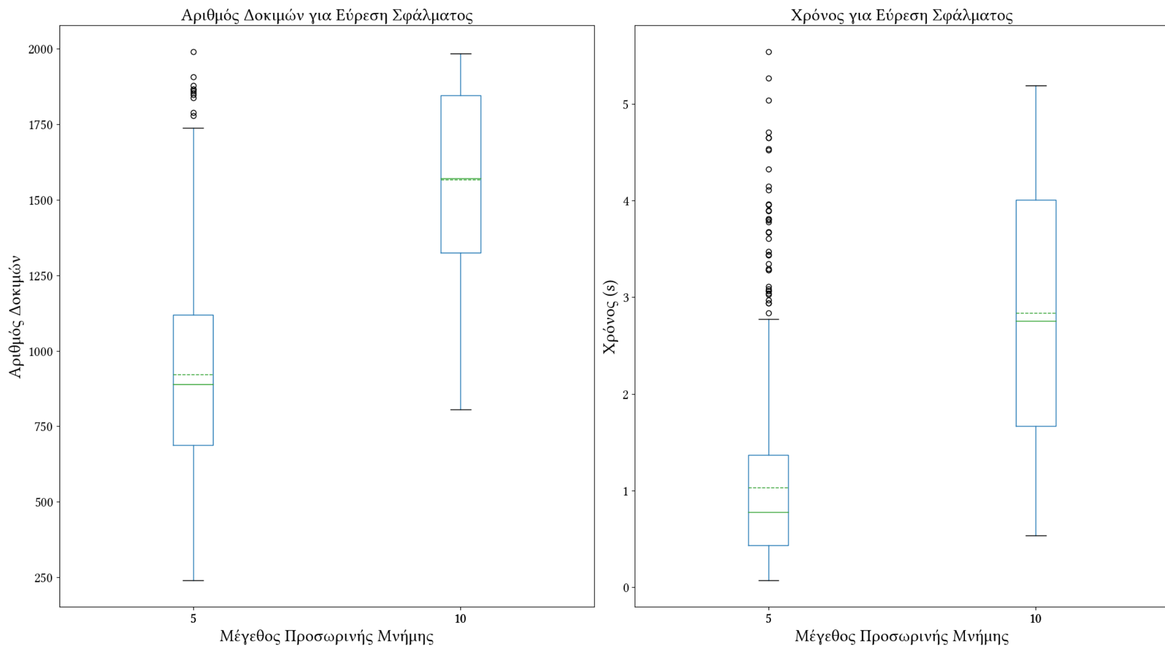
Σχήμα 5.8: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (200 Δοκιμές)



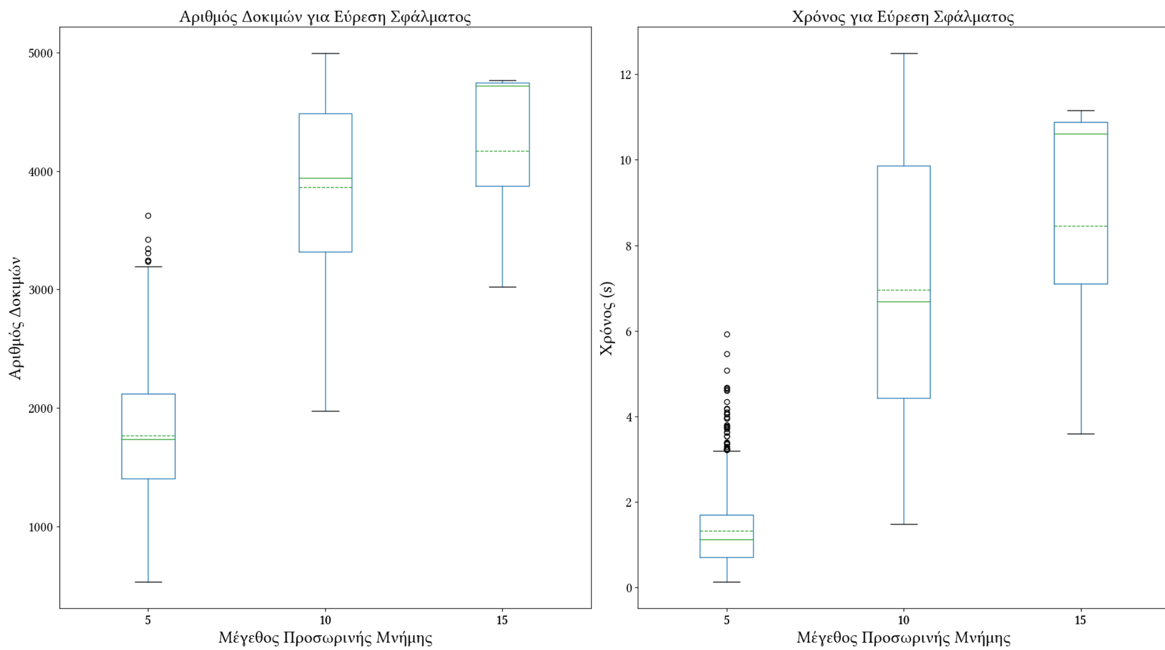
Σχήμα 5.9: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (500 Δοκιμές)



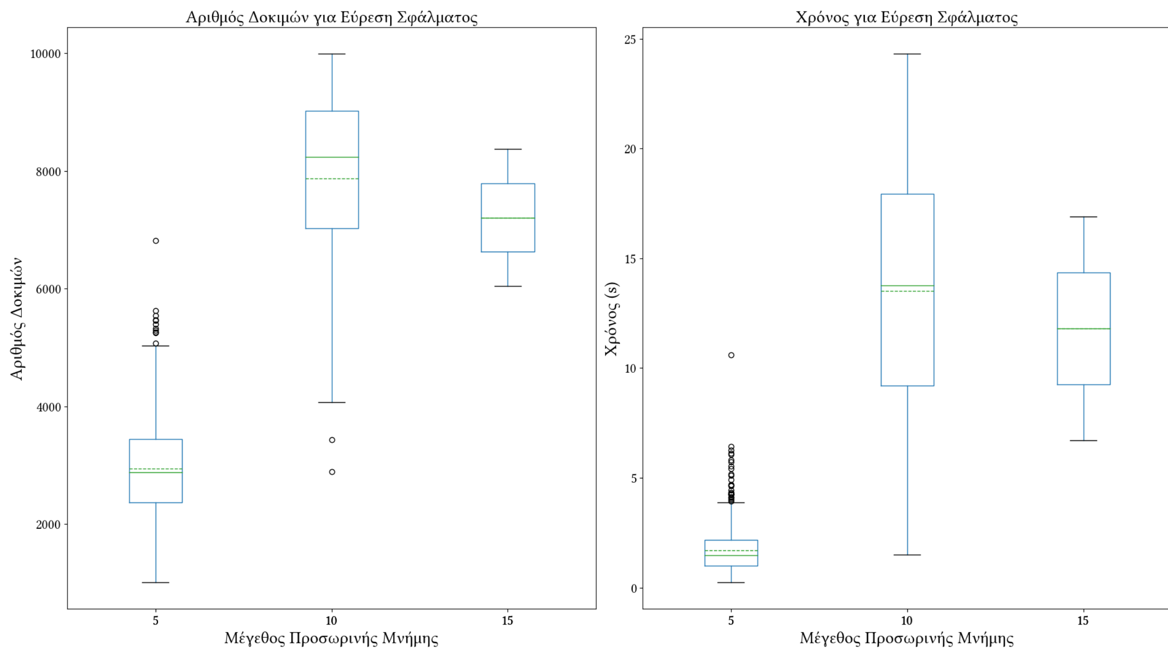
Σχήμα 5.10: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (1000 Δοκιμές)



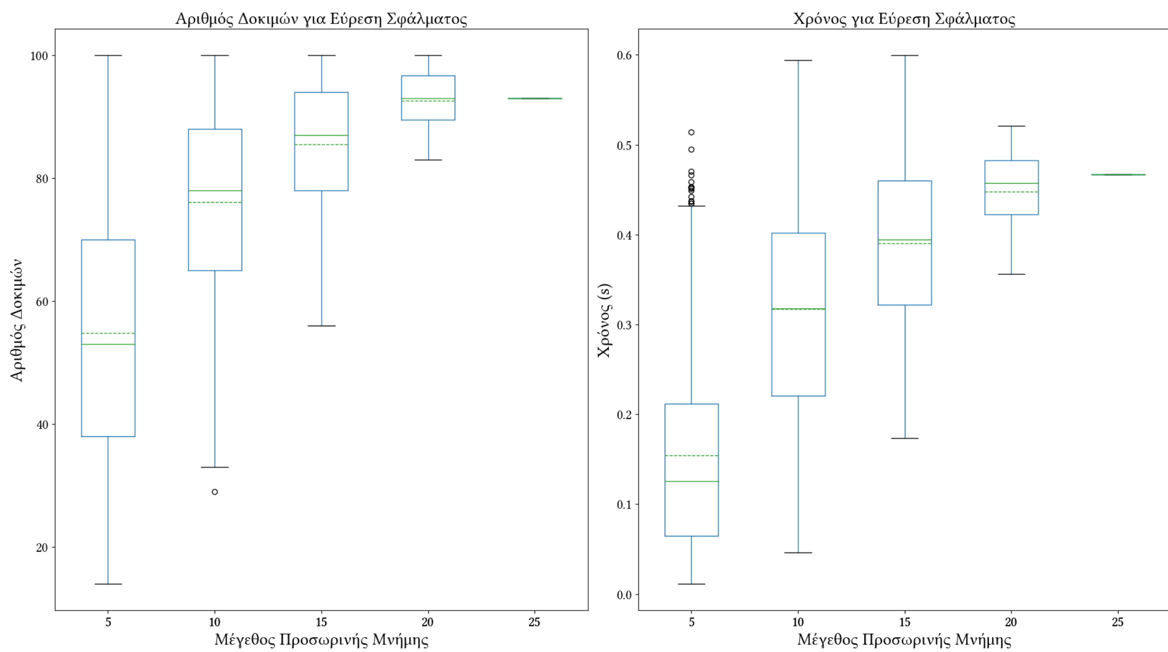
Σχήμα 5.11: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (2000 Δοκιμές)



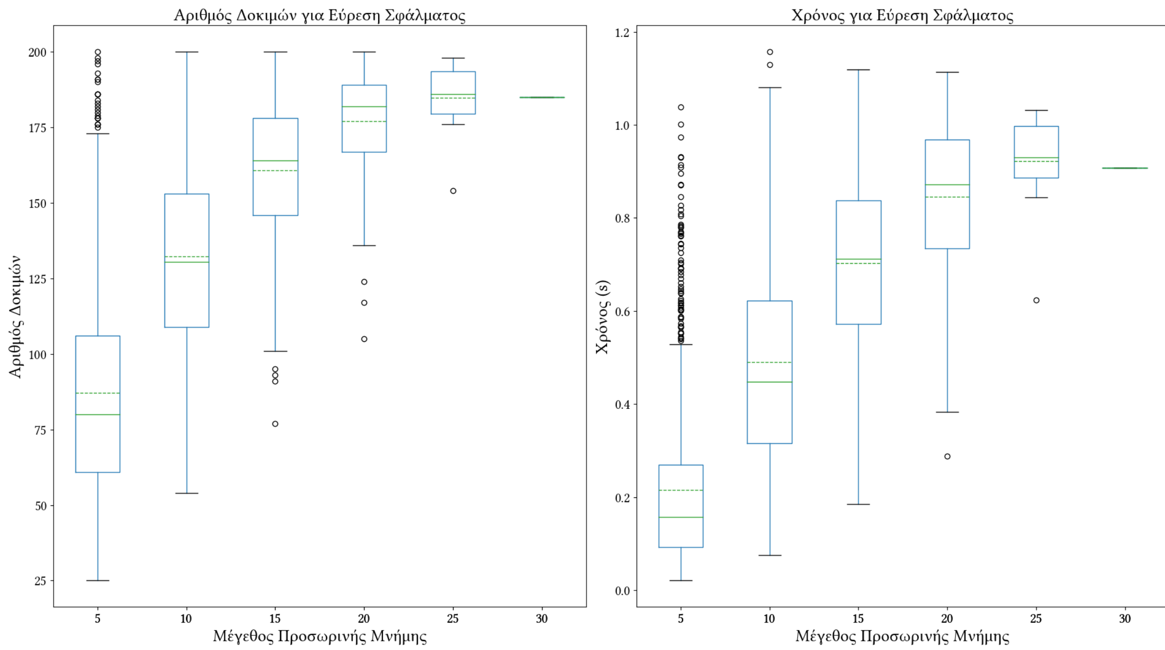
Σχήμα 5.12: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (5000 Δοκιμές)



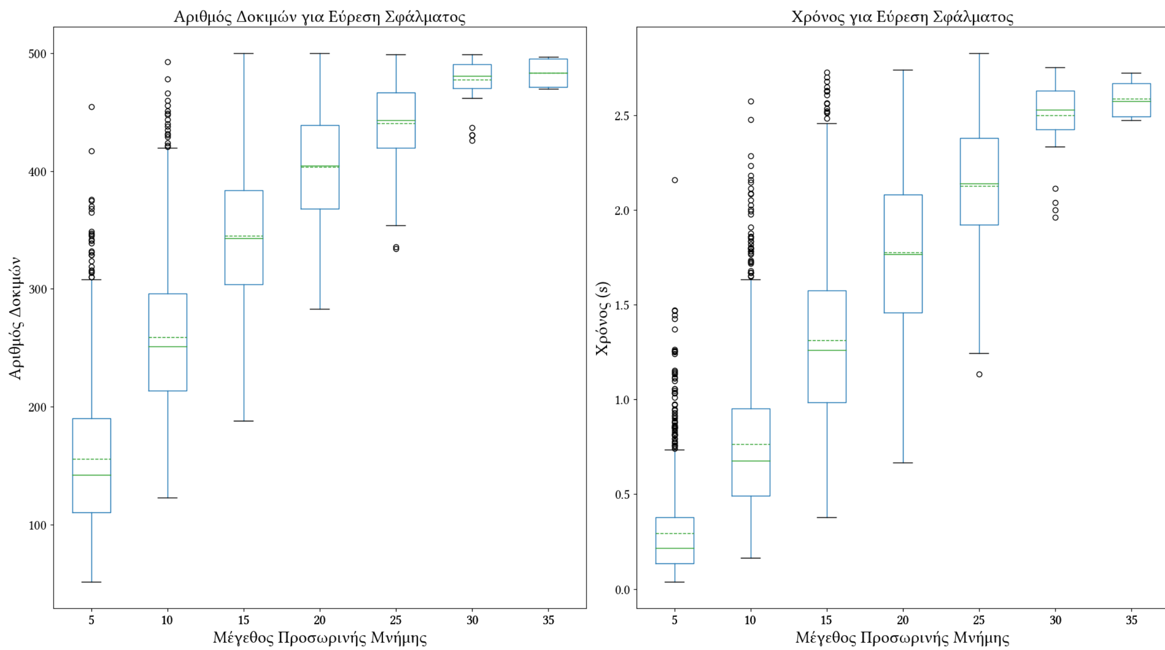
Σχήμα 5.13: Στατιστικά Εύρεσης Σφάλματος στον Τυχαίο Έλεγχο (10000 Δοκιμές)



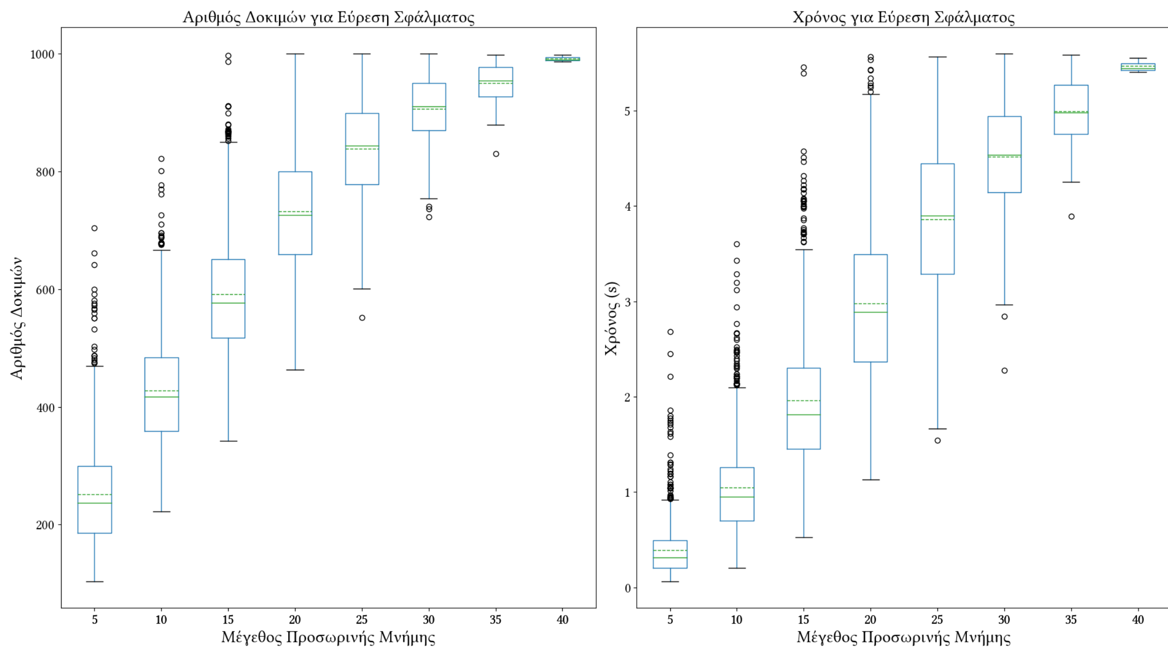
Σχήμα 5.14: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (100 Δοκιμές)



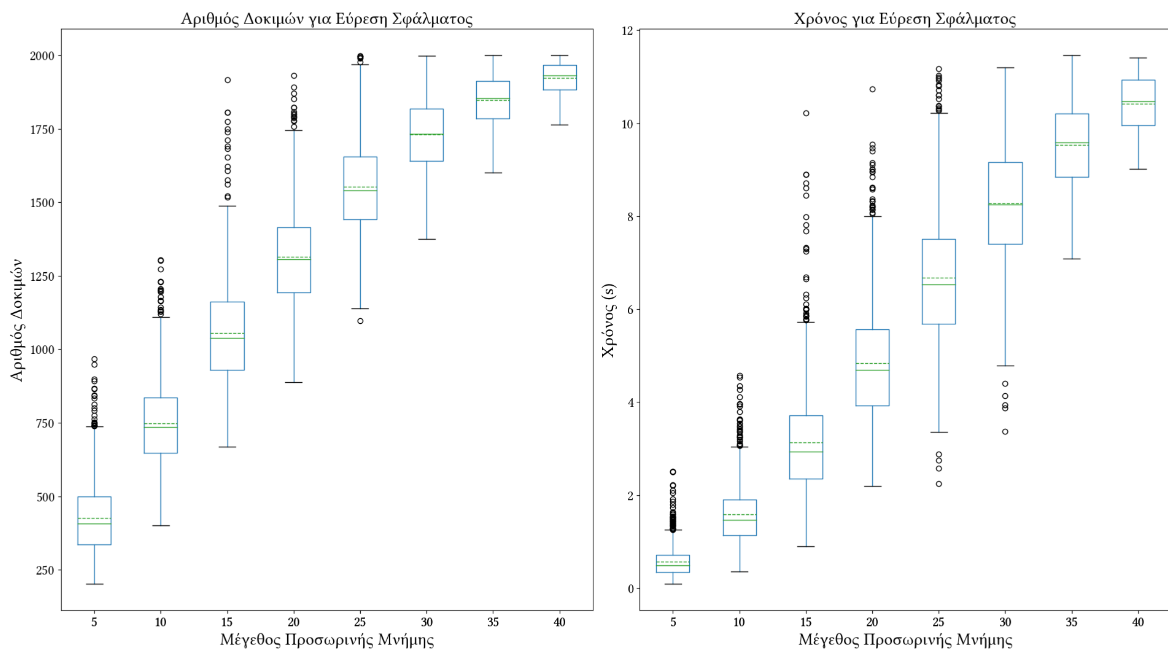
Σχήμα 5.15: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (200 Δοκιμές)



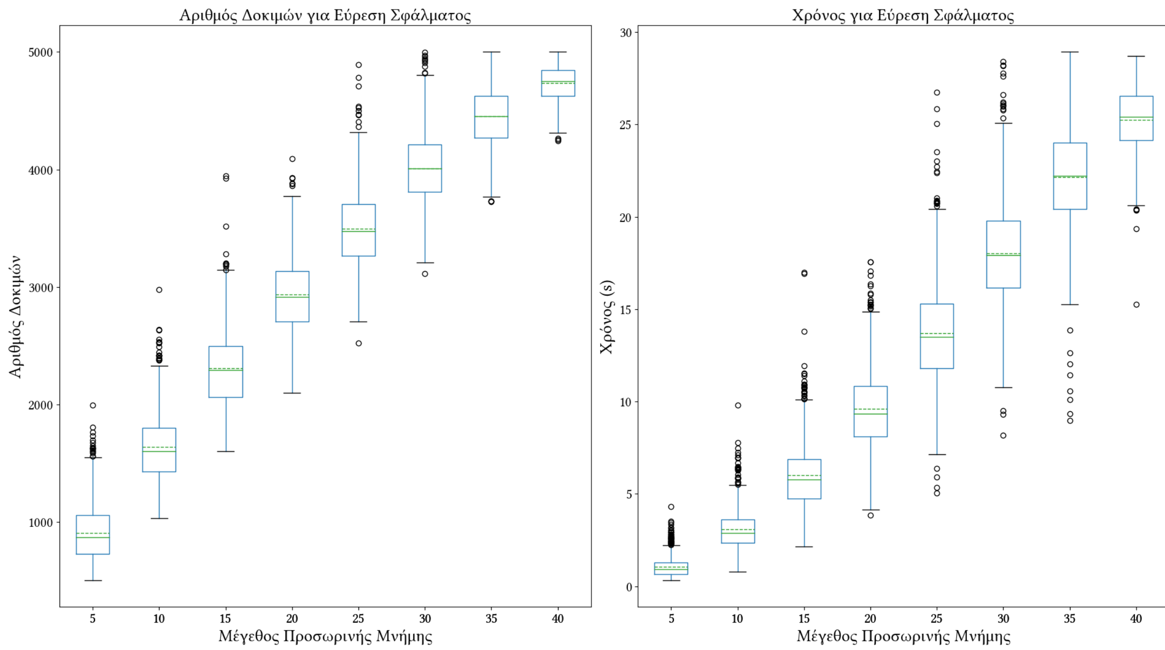
Σχήμα 5.16: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (500 Δοκιμές)



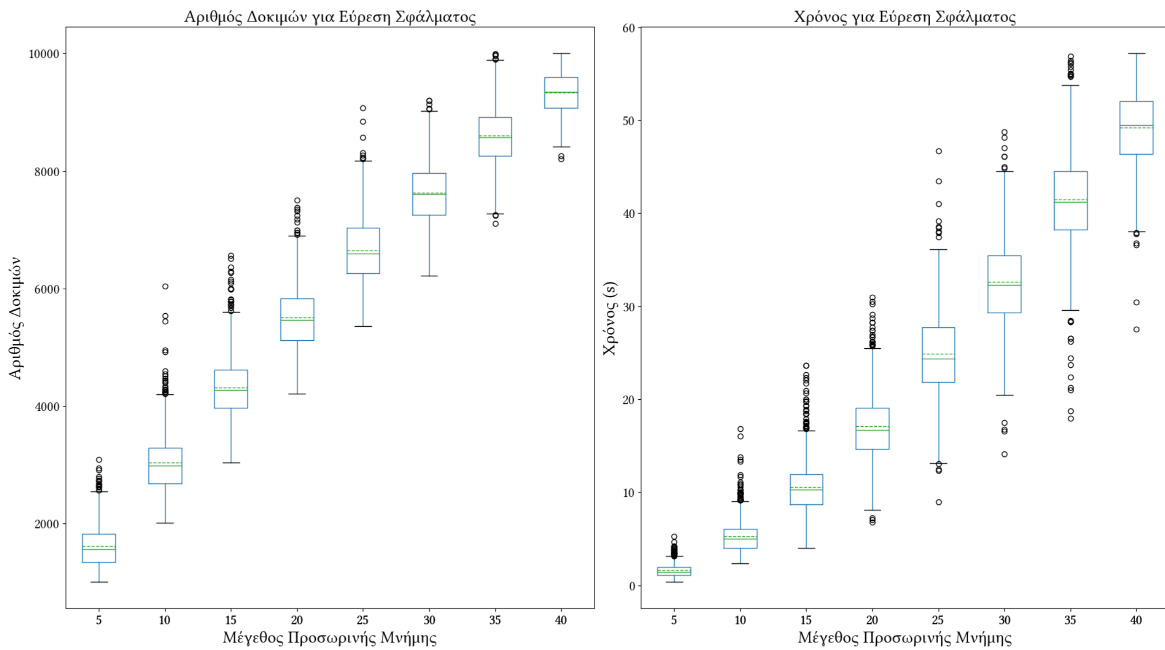
Σχήμα 5.17: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (1000 Δοκιμές)



Σχήμα 5.18: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (2000 Δοκιμές)



Σχήμα 5.19: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (5000 Δοκιμές)



Σχήμα 5.20: Στατιστικά Εύρεσης Σφάλματος στον Στοχευμένο Έλεγχο (10000 Δοκιμές)

5.3 Συμπεράσματα

Κλείνοντας το κεφάλαιο της πειραματικής αξιολόγησης, μπορούμε να συμπεράνουμε ότι η χρήση του στοχευμένου ελέγχου λύνει τα χέρια του προγραμματιστή, ώστε να μπορεί να πετύχει κάτι που χειροκίνητα θα ήθελε περισσότερη δουλειά, με μόλις λίγες επιπλέον γραμμές κώδικα. Ωστόσο, όπως δείξαμε και στα διαγράμματα αυτό δε σημαίνει ότι η διαδικασία είναι τέλεια. Κάθε πρόβλημα είναι ξεχωριστό και πρέπει να εξετάζεται από μόνο του. Στο δικό μας παράδειγμα, είδαμε πως η μεταβολή των παραμέτρων μπορεί να επηρεάσει ραγδαία την αποτελεσματικότητα του PropEr στην εύρεση του σφάλματος, ενώ υπάρχουν και επιπλέον παράμετροι του PropEr που θα πρέπει να λαμβάνονται υπόψιν όταν χρησιμοποιείται ο στοχευμένος έλεγχος. Αυτές υπάρχουν ώστε να μπορεί ο χρήστης να επηρεάσει την εσωτερική λειτουργία του PropEr και να το φέρει "στα μέτρα" του εκάστοτε προβλήματος και καλό είναι να εξετάζονται και αυτές.

Κεφάλαιο 6

Επίλογος

Στο πλαίσιο της παρούσας διπλωματικής, επεκτείναμε το εργαλείο PropEr και τις βιβλιοθήκες του για έλεγχο συστημάτων με κατάσταση, ώστε να μπορεί να πραγματοποιήσει στοχευμένο έλεγχο βάσει ιδιοτήτων σε αυτά. Η συνεισφορά μας επαφίεται στο ότι πρόκειται για ανοιχτό λογισμικό (open source software), το οποίο παρέχεται στο ευρύ κοινό για χρήση ή ακόμα και μετεξέλιξη. Επίσης, παρείχαμε κάποια ενθαρρυντικά στοιχεία μέσω του παραδείγματος με την προσωρινή μνήμη, όμως είναι νωρίς για να βγάλουμε συμπεράσματα για την αποτελεσματικότητα της τεχνικής. Το PropEr, καθώς και οι βιβλιοθήκες του για έλεγχο τέτοιων συστημάτων, χρησιμοποιείται ευρέως τόσο στο Erlang/OTP, όσο και σε άλλες βιβλιοθήκες, οι οποίες θα μπορούσαν να μελετηθούν για την αξιοποίηση της νέας δυνατότητας που εισάγαμε.

6.1 Παρόμοια Έργα

Αξίζει να αναφέρουμε ο τρόπος με τον οποίο χρησιμοποιήσαμε την προσομοίωση ανόπτησης για τον έλεγχο λογισμικού δεν είναι ακριβώς καινούρια τεχνική. Ο έλεγχος βάσει αναζήτησης (ή search-based testing), υπάρχει ήδη σαν όρος από το 1976, ενώ από τότε, τέτοιες τεχνικές ελέγχου έχουν χρησιμοποιηθεί σε διάφορα είδη και εργαλεία του κλάδου [3, 9], αξιοποιώντας μετα-ευριστικές τεχνικές όπως γενετικούς αλγορίθμους, προσομοίωση ανόπτησης, κλπ. Ωστόσο, με βάση τα όσα γνωρίζουμε, τόσο η ενσωμάτωση αυτής της τεχνικής σε ένα εργαλείο ελέγχου βάσει ιδιοτήτων, όσο και η επέκταση για τη χρήση σε συστήματα με κατάσταση αποτελεί την πρώτη στον κλάδο. Αξίζει, επίσης, να αναφερθεί ότι το εργαλείο `Hypothesis`, το οποίο είναι ένα εργαλείο ελέγχου βάσει ιδιοτήτων γραμμένο σε `Python`, χρησιμοποίησε τη λογική στην οποία βασίστηκε το PropEr, ώστε να ενσωματώσει το στοχευμένο έλεγχο στο API που προσφέρει στους χρήστες του [8].

Υπάρχουν κι άλλες τεχνικές που βοηθούν στη δημιουργία όχι εντελώς τυχαίων εισόδων για ένα πρόβλημα, όπως δημιουργία δεδομένων με βάση τη στένωση, τον προσαρμοστικό τυχαίο έλεγχο, τον περιορισμένο τυχαίο έλεγχο ακόμα και η σύνθεση προγραμμάτων από έναν ορισμό υψηλού επιπέδου, κλπ. οι οποίες σίγουρα μπορούν να αποτελέσουν και έμπνευση για τη μελλοντική μας δουλειά.

6.2 Μελλοντική Δουλειά

Στο μέλλον, σκοπεύουμε στη βελτιστοποίηση της τεχνικής του στοχευμένου ελέγχου βάσει ιδιοτήτων, καθώς και στην επέκταση του PropEr ώστε να εκτελεί δοκιμές παράλληλα, δίνοντας τη δυνατότητα για περισσότερες δοκιμές στο ίδιο χρονικό διάστημα. Επίσης, θα θέλαμε να αξιολογήσουμε περαιτέρω τον στοχευμένο έλεγχο βάσει ιδιοτήτων,

χρησιμοποιώντας ήδη υπάρχοντα συστήματα ή πρωτόκολλα, ώστε να μπορούμε να βγάλουμε σαφή συμπεράσματα, καθώς και να βελτιώσουμε την τεχνική όπου χρειάζεται.

Βιβλιογραφία

[1] Eirini Arvaniti. Stateful property based testing, June 2011. URL http://artemis.cslab.ece.ntua.gr:8080/jspui/handle/123456789/16041.

[2] Eirini Arvaniti and Konstantinos Sagonas. A proper state machine tutorial, April 2018. URL https://proper-testing.github.io/tutorials/PropEr_testing_of_generic_servers.html.

[3] Evosuite. Automatic test suite generation for java. URL https://www.evosuite.org/.

[4] Fred Herbert. *Property-Based Testing with PropEr, Erlang, and Elixir*. The Pragmatic Programmers, 2019. ISBN 978-1-68050-621-1.

[5] Andreas Löscher and Konstantinos Sagonas. Targeted property-based testing. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2017, pages 46–56, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5076-1. doi: 10.1145/3092703.3092711. URL http://doi.acm.org/10.1145/3092703.3092711.

[6] Andreas Löscher and Konstantinos Sagonas. Tutorial for targeted property-based testing, April 2018. URL https://proper-testing.github.io/tutorials/PropEr_testing_with_search_strategies.html.

[7] Andreas Löscher and Konstantinos Sagonas. Automating targeted property-based testing. In *11th IEEE International Conference on Software Testing, Verification and Validation*, ICST 2018, pages 70–80. IEEE Computer Society, April 2018. doi: 10.1109/ICST.2018.00017. URL https://doi.org/10.1109/ICST.2018.00017.

[8] David R. MacIver. Hypothesis advanced features and examples, 2020. URL https://hypothesis.readthedocs.io/en/latest/details.html#targeted-example-generation.

[9] Michael Mayo and Simon Spacey. Predicting regression test failures using genetic algorithm-selected dynamic performance analysis metrics. In Günther Ruhe and Yuanyuan Zhang, editors, *Search Based Software Engineering*, pages 158–171, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi: 10.1007/978-3-642-39742-4_13. URL https://link.springer.com/chapter/10.1007/978-3-642-39742-4_13.

[10] Alexander G. Nikolaev and Sheldon H. Jacobson. Simulated annealing. In *Handbook of Metaheuristics*, pages 1–39. Springer, 2010.

[11] Manolis Papadakis and Konstantinos Sagonas. A PropEr integration of types and function specifications with property-based testing. In *Proceedings of the 10th ACM*

SIGPLAN Workshop on Erlang, Erlang '11, pages 39–50, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0859-5. doi: 10.1145/2034654.2034663. URL http://doi.acm.org/10.1145/2034654.2034663.

[12] Andreas Zeller and Ralf Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Trans. Softw. Eng.*, 28(2):183–200, February 2002. ISSN 0098-5589. doi: 10.1109/32.988498. URL https://doi.org/10.1109/32.988498.