Ατομική Διπλωματική Εργασία

ΈΛΕΓΧΟΣ ΛΟΓΙΣΜΙΚΟΥ ΜΕ ΤΗ ΧΡΗΣΗ ΜΟΝΤΕΛΩΝ : ΘΕΩΡΗΤΙΚΗ ΠΡΟΣΑΡΜΟΓΗ ΑΚΟΛΟΥΘΙΑΚΩΝ ΔΙΑΓΡΑΜΜΑΤΩΝ

Ευγενία Βασιλείου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάϊος 2009

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Έλεγχος λογισμικού με τη χρήση μοντέλων : θεωρητική προσαρμογή ακολουθιακών διαγραμμάτων

Ευγενία Βασιλείου

Επιβλέπων Καθηγητής
Ανδρέου Ανδρέας

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

Μάϊος 2009

Περίληψη

Το model-checking, δηλαδή ο έλεγχος κάποιου συστήματος με τη χρήση μοντέλων, είναι μια πολύ χρήσιμη διαδικασία στην ανάπτυξη λογισμικού. Όταν τα λάθη βρίσκονται αργά, τότε το κόστος του να τα διορθώσουμε είναι πολύ μεγαλύτερο από αυτό του να τα διορθώσουμε έγκαιρα.

Υπάρχουν πολλά μοντέλα για να κάνουμε model-checking σε ένα σύστημα, όπως για παράδειγμα οι πίνακες απόφασης, οι μηχανές πεπερασμένων καταστάσεων, οι γραμματικές, οι αλυσίδες Markov και τα UML.

Χρησιμοποιώντας διαγράμματα ακολουθιών για να ελέγξουμε ένα πρόγραμμα στην Java, ανακαλύψαμε ότι υπάρχουν πράγματα τα οποία δεν υποστηρίζουν τα διαγράμματα κι έτσι ο έλεγχος δεν μπορεί να γίνει ολοκληρωμένα. Λόγω του ότι το model-checking χρησιμοποιεί μοντέλα τα οποία είναι ένα στιγμιότυπο του συστήματος και ελέγχονται εκείνα αντί για το σύστημα, ο έλεγχος είναι όσο καλός όσο είναι και τα μοντέλα τα οποία χρησιμοποιήθηκαν. Γι' αυτό έπρεπε να εντοπιστούν τα προβλήματα αυτά στα διαγράμματα ακολουθίας και στη συνέχεια πρότεινα λύσεις για τη βελτίωσή τους έτσι ώστε να γίνεται ένας πιο ολοκληρωμένος έλεγχος του μοντέλου και κατά συνέπεια του συστήματος.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Τι είναι ένα μοντέλο

Για να αρχίσουμε να μιλάμε για model-checking πρέπει πρώτα να αναφέρουμε το τι είναι μοντέλο. Ένα μοντέλο του λογισμικού είναι μια απεικόνιση της συμπεριφοράς του. Η συμπεριφορά μπορεί να περιγραφεί από την άποψη των ακολουθιών εισαγωγής που γίνονται αποδεκτές από το σύστημα, τις ενέργειες, τις συνθήκες και την λογική των εξόδων (output) ή τη ροή των στοιχείων μέσω των ενοτήτων της εφαρμογής.

Υπάρχουν πολυάριθμα τέτοια μοντέλα και καθένα περιγράφει τις διαφορετικές πτυχές της συμπεριφοράς λογισμικού. Στο Κεφάλαιο 3 θα εξετάσουμε κάποια από αυτά.

Κεφάλαιο 2

Model-Checking

2.1 Τι είναι το model-checking

Ο ορισμός του model-checking :

Το model-checking είναι μια αυτοματοποιημένη τεχνική η οποία δεδομένου ενός πεπερασμένου αριθμού καταστάσεων μοντέλο ενός συστήματος και μίας τυπικής ιδιότητας, ελέγχει συστηματικά αν αυτή η ιδιότητα ευσταθεί στο μοντέλο [1].

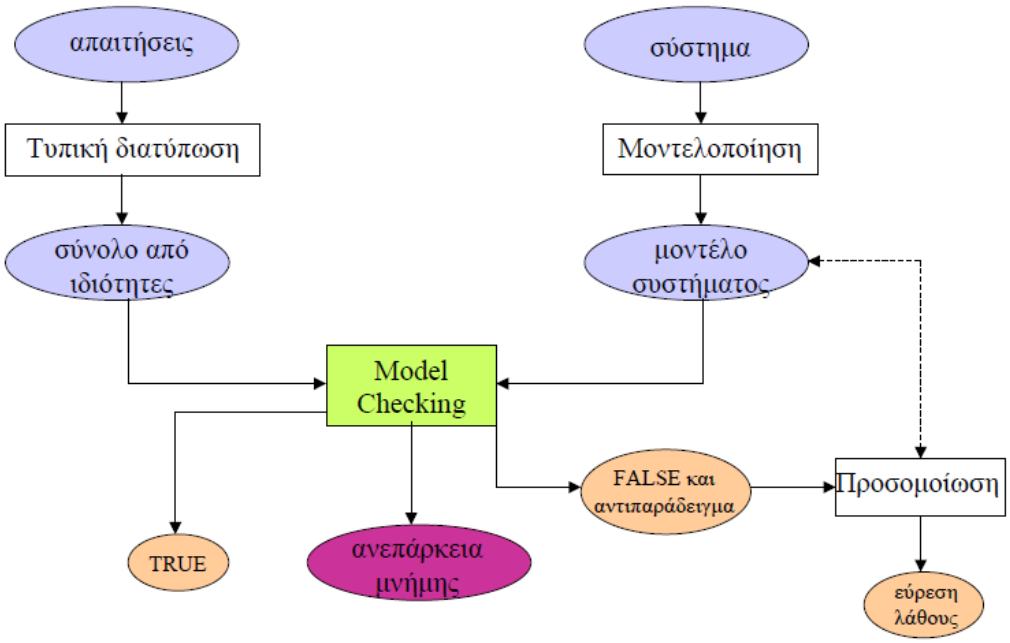
Στο σχεδιασμό λογισμικού και υλικού πολυσύνθετων συστημάτων, ο περισσότερος χρόνος και η περισσότερη προσπάθεια ξοδεύονται στην επαλήθευση παρ' ότι στην κατασκευή. Οι τεχνικές επιδιώκουν να μειώσουν και να διευκολύνουν τις προσπάθειες επαλήθευσης αυξάνοντας την κάλυψή τους. Οι τυπικές μέθοδοι προσφέρουν μια μεγάλη δυνατότητα να ληφθεί μια πρόωρη ενσωμάτωση της επαλήθευσης στη διαδικασία σχεδίου, να παρασχεθούν οι αποτελεσματικότερες τεχνικές επαλήθευσης, και να μειωθεί ο χρόνος επαλήθευσης.

Οι βασισμένες σε μοντέλα τεχνικές επαλήθευσης (model-based verification techniques) είναι βασισμένες σε μοντέλα που περιγράφουν την πιθανή συμπεριφορά του

συστήματος με τρόπο μαθηματικά ακριβή και σαφή. Πριν από οποιαδήποτε μορφή επαλήθευσης, η ακριβής μοντελοποίηση των συστημάτων οδηγεί συχνά στην ανακάλυψη της ημιτέλειας, των ασαφειών και των ασυνεπειών στις άτυπες προδιαγραφές συστημάτων. Τέτοια προβλήματα συνήθως ανακαλύπτονται σε ένα πολύ προχωρημένο στάδιο του σχεδιασμού. Τα μοντέλα συστημάτων συνοδεύονται από τους αλγορίθμους που ερευνούν συστηματικά όλες τις καταστάσεις τους. Αυτό παρέχει τη βάση για μια ολόκληρη σειρά τεχνικών επαλήθευσης που κυμαίνονται από εξαντλητική εξερεύνηση (model checking) μέχρι και τα πειράματα με ένα περιορισμένο σύνολο σεναρίων στο μοντέλο (προσομοίωση) ή στην πραγματικότητα (testing). Λόγω των επίμονων βελτιώσεων των υποκρυπτόμενων αλγορίθμων και των δομών δεδομένων, μαζί με τη διαθεσιμότητα των γρηγορότερων υπολογιστών και των μεγαλύτερων μνημών υπολογιστών, οι βασισμένες σε μοντέλα τεχνικές για τις οποίες πριν μια δεκαετία εφαρμόζονταν μόνο σε πολύ απλά παραδείγματα είναι σήμερα εφαρμόσιμες σε ρεαλιστικά σχεδιασμούς. Λόγω του ότι σαν αφετηρία αυτών των τεχνικών είναι ένα μοντέλο του συστήματος υπό εξέταση, έχουμε ως δεδομένο ότι :

Οποιαδήποτε επαλήθευση χρησιμοποιεί τεχνικές βασισμένες σε μοντέλα
είναι όσο καλή όσο το μοντέλο του συστήματος. [1]

Το model-checking είναι μια τεχνική επαλήθευσης που ερευνά όλες τις πιθανές καταστάσεις του συστήματος με brute-force τρόπο. Το εργαλείο λογισμικού που εκτελεί το model-checking ερευνά όλα τα πιθανά σενάρια με μεθοδικό τρόπο. Έτσι, μπορεί να δείξει ότι ένα δεδομένο μοντέλο συστήματος ικανοποιεί μια συγκεκριμένη ιδιότητα. Χρησιμοποιώντας έξυπνους αλγόριθμους και εφαρμοσμένες δομές δεδομένων μπορούν να χειριστούν χώροι μεγάλων καταστάσεων για συγκεκριμένα προβλήματα (από 10^{20} μέχρι και 10^{476} καταστάσεις [3]). Ακόμα και τα δυσδιάκριτα λάθη που δεν ανακαλύπτονται, μπορούν πιθανώς να αποκαλυφθούν με τη χρήση του model-checking.



Σχήμα 2.1 : Σχηματική αναπαράσταση της προσέγγισης με model-checking [2]

Οι τυπικές ιδιότητες που μπορούν να ελεγχθούν χρησιμοποιώντας model-checking είναι ποιοτικής φύσεως : Είναι το παραγόμενο αποτέλεσμα σωστό; Μπορεί το σύστημα να φτάσει σε αδιέξοδο; Αλλά μπορούν επίσης να ελεγχθούν και χρονικές ιδιότητες : Μια απάντηση λαμβάνεται εντός 5 λεπτών; Το model-checking απαιτεί μια ακριβή δήλωση των ιδιοτήτων που θα ελεγχθούν. Αυτό το βήμα συχνά οδηγεί στην ανακάλυψη ασαφειών και ανακολουθιών.

Το μοντέλο του συστήματος παράγεται συνήθως αυτόματα από μια περιγραφή του μοντέλου που είναι ορισμένη σε μια κατάλληλη γλώσσα προγραμματισμού όπως είναι η C ή η Java ή γλώσσες περιγραφής του υλικού (hardware) όπως η Verilog ή η VHDL. Ο προσδιορισμός των ιδιοτήτων υπαγορεύει το τι πρέπει να κάνει το σύστημα και τι δεν πρέπει να κάνει, ενώ η περιγραφή του μοντέλου δείχνει το πώς συμπεριφέρεται το σύστημα. Ο ελεγκτής του μοντέλου (model checker) εξετάζει όλες τις σχετικές καταστάσεις του συστήματος για να ελέγξει αν ικανοποιούν την επιθυμητή ιδιότητα. Αν μια κατάσταση προσεγγιστεί η οποία παραβιάζει την υπό εξέταση ιδιότητα, ο ελεγκτής του μοντέλου παρέχει ένα παράδειγμα (counterexample) που καθορίζει πώς το μοντέλο θα μπορούσε να φτάσει στην ανεπιθύμητη κατάσταση.

Η δοκιμή περιγράφει το μονοπάτι εκτέλεσης που οδηγεί από την αρχική κατάσταση του συστήματος σε μια κατάσταση που παραβιάζει την ιδιότητα που επαληθεύεται. Με τη βοήθεια ενός προσομοιωτή, ο χρήστης μπορεί να επαναλάβει το αθέμιτο σενάριο, αποκτώντας έτσι χρήσιμες πληροφορίες για αποσφαλμάτωση (debugging) και να προσαρμόσει το μοντέλο κατάλληλα.

2.2 Ιστορία του model-checking

Το model-checking προέρχεται από την ανεξάρτητη δουλειά δύο ζευγαριών στην αρχή της δεκαετίας του ’80 : των E.M. Clarke και E.A. Emerson και των J. Sifakis και. J. P. Queille. Οι τρεις πρώτοι μοιράστηκαν το βραβείο Turing το 2007 για τη δουλειά τους πάνω στο model-checking. [5]

Ο όρος model-checking προτάθηκε από τους Clarke και Emerson. Η brute-force εξέταση όλου του χώρου καταστάσεων στο model-checking μπορεί να θεωρηθεί ως επέκταση των αυτοματοποιημένων τεχνικών επικύρωσης πρωτόκολλου από τους Hajek και West. Οι περιορισμοί του model-checking συζητήθηκαν από τους Apt και Kozen. [5]

Περισσότερες πληροφορίες για το model-checking είναι διαθέσιμες στα αρχικά βιβλία των Holzmann, McMillan και Kurshan και την πιο πρόσφατη δουλειά των Clarke, Grumberg και Peled και των Huth, Ryan Schneider και Berard. Η τροχιά του model-checking περιγράφηκε πρόσφατα από τους Ruys και Brinksma. [5]

2.3 Επαλήθευση συστημάτων

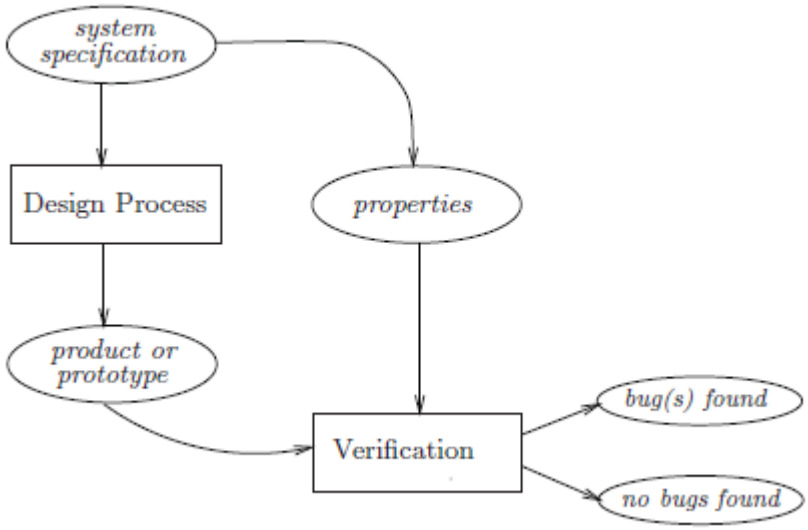
Σε αυτό το υποκεφάλαιο θα δούμε γιατί το model-checking είναι χρήσιμο στην ανάπτυξη λογισμικού και υλικού.

Η εμπιστοσύνη μας στη λειτουργία των συστημάτων πληροφοριών και επικοινωνιακών τεχνολογιών (Information and Communication Technology – ICT) μεγαλώνει ραγδαία. Τα συστήματα αυτά γίνονται όλο και πιο περίπλοκα και διεισδύουν μαζικά στην καθημερινή μας ζωή μέσω του διαδικτύου και όλων των ειδών ενσωματωμένων συστημάτων όπως οι smart cards, οι φορητοί υπολογιστές, τα κινητά κτλ. Το 1995 υπολογίστηκε ότι χρησιμοποιούμε περίπου 25 τέτοιες συσκευές επί καθημερινής βάσεως [1]. Υπηρεσίες όπως ηλεκτρονικές τραπεζικές συναλλαγές και αγορές μέσω του διαδικτύου έχουν γίνει πραγματικότητα. Τα συστήματα ICT είναι παγκόσμια και βρίσκονται παντού. Ελέγχουν το χρηματιστήριο, αποτελούν την «καρδιά» των τηλεφωνικών διακόπτων, είναι κρίσιμα για το διαδίκτυο και είναι απαραίτητα για πολλά συστήματα ιατρικής. Η εμπιστοσύνη μας σε αυτά τα συστήματα κάνει την αξιόπιστη λειτουργία τους μεγάλης κοινωνικής σημασίας. Εκτός του να προσφέρουν καλή απόδοση σε θέματα χρονικής σημασίας, η απουσία ενοχλητικών λαθών είναι μια από τις μεγάλες ενδείξεις ποιότητας.

Κάθε φορά που κάποια συσκευή που χρησιμοποιούμε δυσλειτουργεί ενοχλούμαστε. Μπορεί αυτές οι δυσλειτουργίες να μην απειλούν τη ζωή μας, αλλά μπορεί να έχουν σημαντικές οικονομικές κυρώσεις για τον κατασκευαστή. Σωστά ICT συστήματα είναι μεγάλης σημασίας για την επιβίωση μιας εταιρείας. Πολλά παραδείγματα είναι γνωστά. Το ελάττωμα στο Pentium II της Intel στη μονάδα τμήματος κινητής υποδιαστολής στην αρχή της δεκαετίας του ’90, προκάλεσε απώλεια περίπου 475 εκατομμυρίων δολλαρίων για την αντικατάσταση των ελαττωματικών επεξεργαστών και ζήμιωσε σημαντικά τη φήμη της Intel σαν αξιόπιστος κατασκευαστής chips. Το λογισμικό λάθος σε ένα σύστημα χειρισμού αποσκευών καθυστέρησε το άνοιγμα του αεροδρομίου του Denver εννιά μήνες και είχαν απώλεια 1,1 εκατομμυρίων δολλαρίων την ημέρα. Τα λάθη μπορούν να είναι όμως και καταστροφικά. Οι ατέλειες στον έλεγχο του λογισμικού του πύραυλου Ariane-5 (του ανιχνευτή του Άρη) το οποίο εκτοξεύθηκε στις 4 Ιουνίου 1996 προκάλεσαν τη συντριβή του 36 δευτερόλεπτα μετά λόγω μιας

λανθασμένης μετατροπής ενός 64-bit floating point σε 16-bit integer. Ένα λάθος λογισμικού στο μέρος ελέγχου της μηχανής ραδιοθεραπείας Therac-5 προκάλεσε τον θάνατο σε έξι ασθενείς καρκίνου σε διάστημα δύο χρόνων αφού εκτέθηκαν σε υπερβολική δόση ραδιενέργειας[1,2].

Οι τεχνικές επαλήθευσης συστημάτων εφαρμόζονται στο σχεδιασμό των ICT συστημάτων με έναν πιο αξιόπιστο τρόπο. Η επαλήθευση συστημάτων χρησιμοποιείται για να εξασφαλίσουμε ότι ο σχεδιασμός ή το προϊόν υπό εξέταση κατέχει κάποιες ιδιότητες. Οι ιδιότητες που τίθονται προς επικύρωση μπορεί να είναι στοιχειώδεις, για παράδειγμα ένα σύστημα δεν πρέπει να φτάσει ποτέ σε μια κατάσταση όπου να μην μπορεί να βγει από αυτή (deadlock). Αυτές συνήθως αποκτούνται από τις προδιαγραφές του συστήματος. Οι προδιαγραφές περιγράφουν το τι πρέπει και τι δεν πρέπει να κάνει το σύστημα κι έτσι αποτελούν τη βάση για την επαλήθευση του συστήματος. Ένα ελάττωμα βρίσκεται όταν το σύστημα δεν ικανοποιεί μία από τις προδιαγραφές. Το σύστημα θεωρείται σωστό όταν ικανοποιεί όλες τις ιδιότητες που αποκτήθηκαν από τις προδιαγραφές.



Σχήμα 2.2 : Σχηματική αναπαράσταση της επαλήθευσης συστημάτων [1]

2.4 Επαλήθευση λογισμικού

Οι τεχνικές που χρησιμοποιούνται κυρίως για επαλήθευση λογισμικού είναι το peer reviewing και οι δοκιμές (testing).

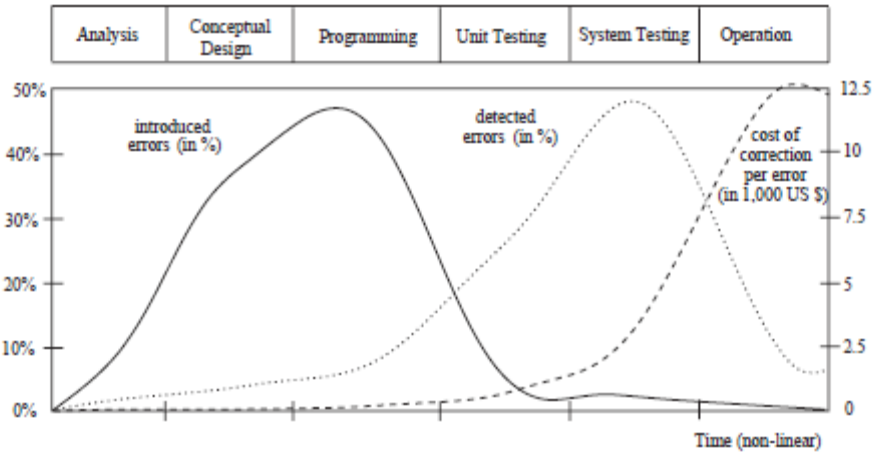
Peer review : το λογισμικό δέχεται επιθεώρηση από μια ομάδα μηχανικών λογισμικού, οι οποίοι κατά προτίμηση δεν συνέβαλαν στη δημιουργία του λογισμικού αυτού. Ο μη μεταγλωττισμένος κώδικας δεν εκτελείται αλλά αναλύεται στατικά. Μελέτες έχουν δείξει ότι το peer review είναι μια αποτελεσματική τεχνική η οποία λαμβάνει το 31% μέχρι και το 93% των λαθών με ένα μέσο όρο 60%. Παρόλο που είναι μια χειρωνακτική εργασία, το peer review είναι μια χρήσιμη τεχνική, γι'αυτό και χρησιμοποιείται στο 80% περίπου των προγραμματιστικών έργων (projects) της τεχνολογίας λογισμικού [3]. Όμως λόγω του ότι είναι στατική διαδικασία, τα δυσδιάκριτα λάθη όπως είναι τα προβλήματα ταυτοχρονίας είναι δύσκολο να ανιχνευθούν με το peer review.

Testing : Αποτελεί ένα σημαντικό μέρος οποιουδήποτε project της τεχνολογίας λογισμικού. Μεταξύ του 30 και του 50% του ολικού κόστους του project αφιερώνεται στο testing [3]. Σε αντίθεση με το peer review που αναλύει τον κώδικα στατικά χωρίς να τον εκτελεί, το testing είναι μια δυναμική τεχνική, δηλαδή εκτελεί τον κώδικα. Πιο συγκεκριμένα, παίρνει το μέρος του λογισμικού υπό εξέταση και δίνει στον εκτελέσιμο κώδικα εισόδους, που ονομάζονται δοκιμές. Η ορθότητα καθορίζεται αναγκάζοντας το λογισμικό να περάσει από κάποια μονοπάτια εκτέλεσης, ακολουθίες δηλώσεων κώδικα που αναπαριστούν ένα τρέξιμο (run) του κώδικα. Βάση των παρατηρήσεων κατά τη διάρκεια των εκτελέσεων των δοκιμών, η πραγματική έξοδος του συστήματος συγκρίνεται με την έξοδο που καταγράφηκε βάση των προδιαγραφών του συστήματος. Η γενίκευση και έκτελεση των δοκιμών γίνονται συνήθως αυτοματοποιημένα, ενώ η σύγκριση γίνεται συνήθως από ανθρώπους. Το κυριότερο προτέρημα του testing είναι ότι μπορεί να εφαρμοστεί σε πολλά είδη λογισμικών, από λογισμικά εφαρμογών μέχρι μεταγλωττιστές και λειτουργικά συστήματα. Το να ελεγχθούν όλα τα μονοπάτια εκτέλεσης του συστήματος είναι πρακτικά αδύνατο, στην πραγματικότητα όμως ένα μικρό υποσύνολο αυτών των μονοπατιών αντιμετωπίζονται. Άρα το testing δεν θα είναι ποτέ ολοκληρωμένο, δηλαδή μπορεί να δείξει την παρουσία λαθών αλλά δεν μπορεί να

δείξει την απουσία τους. Ακόμα ένα πρόβλημα είναι ο καθορισμός του πότε πρέπει να σταματήσει.

Έρευνες έχουν δείξει ότι οι δύο τεχνικές συλλαμβάνουν διαφορετικές κλάσεις λαθών σε διαφορετικά στάδια του κύκλου ανάπτυξης. Γι' αυτό και συχνά χρησιμοποιούνται μαζί. Για να αυξηθεί η αξιοπιστία του λογισμικού, αυτές οι τεχνικές επαλήθευσης συμπληρώνονται με άλλες τεχνικές όπως δομημένες μέθοδοι σχεδίου και προδιαγραφών (π.χ. UML). Οι τυπικές τεχνικές χρησιμοποιούνται στο 10 με 15% όλων των project λογισμικών.

Λήψη λαθών λογισμικού : όσο πιο γρήγορα τόσο το καλύτερο. Είναι μεγάλης σημασίας να εντοπιστούν τα λάθη του λογισμικού. Το κόστος του να διορθωθεί ένα ελάττωμα στο λογισμικό κατά τη διάρκεια της συντήρησης είναι περίπου 500 φορές μεγαλύτερο από το να διορθωθεί στα αρχικά στάδια του σχεδιασμού. Άρα το σύστημα επαλήθευσης πρέπει να γίνει στα αρχικά στάδια της διαδικασίας σχεδιασμού.



Σχήμα 2.3 : Παρουσίαση λαθών, εντοπισμός και κόστος διόρθωσης [1]

Όπως παρατηρούμε από την πιο πάνω γραφική παράσταση περίπου το 50% όλων των λαθών παρουσιάζονται κατά τη διάρκεια του προγραμματισμού, δηλαδή στη φάση όπου γράφεται ο κώδικας. Παρόλο που μόνο 15% όλων των λαθών παρουσιάζονται στα αρχικά στάδια του σχεδιασμού, τα περισσότερα λάθη βρίσκονται κατά τη διάρκεια

του testing. Στην αρχή του testing, το οποίο προσανατολίζεται στο να βρει ελαττώματα στις ανεξάρτητες μονάδες του λογισμικού που απαρτίζουν το σύστημα, η πυκνότητα των ελαττωμάτων είναι περίπου 20 ανά 1000 γραμμές κώδικα (χωρίς σχόλια). Αυτό έχει μειωθεί σε περίπου 6 ελαττώματα ανά 1000 γραμμές κώδικα στην αρχή του testing του συστήματος. Για να κυκλοφορήσει ένα νέο λογισμικό, ο ανεκτός αριθμός ελαττωμάτων είναι συνήθως 1 ανά 1000 γραμμές κώδικα.

Τα λάθη συνήθως επικεντρώνονται σε μερικά μέρη του λογισμικού. Περίπου τα μισά μέρη δεν έχουν λάθη και περίπου το 80% των λαθών βρίσκονται σε ένα μικρό ποσοστό των μερών του λογισμικού (περίπου στο 20%). Η διόρθωση των λαθών που βρίσκονται πριν από το testing γίνεται αρκετά οικονομικά. Το κόστος της διόρθωσης όπως βλέπουμε στην πιο πάνω γραφική παράσταση αυξάνεται σημαντικά από περίπου $1000 (ανά διόρθωση λάθους) στη φάση του testing σε περίπου $12500 όταν το λάθος παρουσιάζεται κατά τη διάρκεια της λειτουργίας του συστήματος. Είναι άρα πολύ σημαντικό να βρεθούν τεχνικές που να εντοπίζουν τα ελαττώματα όσο το γρηγορότερο δυνατό στη φάση σχεδιασμού του λογισμικού γιατί το κόστος της διόρθωσης είναι σημαντικά πιο χαμηλό και η επιρροή τους στον υπόλοιπο σχεδιασμό είναι λιγότερο ουσιαστική.

2.5 Επαλήθευση υλικού

Το να αποτραπούν τα λάθη στο σχεδιασμό υλικού είναι κρίσιμο γιατί το υλικό υπόκειται σε ψηλά κόστα επεξεργασίας. Το να διορθωθούν τα ελαττώματα μετά την παράδοσή του στους πελάτες είναι πολύ δύσκολο και οι απαιτήσεις ποιότητας είναι πολύ ψηλές. Ενώ τα λάθη του λογισμικού μπορούν να διορθωθούν με το να παρέχεται στους χρήστες αναβάθμιση, τα ελαττώματα του υλικού είναι δύσκολο να διορθωθούν μετά την παράδοσή του στους πελάτες και χρειάζονται κυρίως να ξαναεπεξεργαστούν και να ξαναδιανομηθούν. Αυτό έχει μεγάλες οικονομικές συνέπειες.

Τεχνικές για επαλήθευση υλικού : προσομοίωση, εξομοίωση και ανάλυση δομών είναι οι κύριες τεχνικές που χρησιμοποιούνται για την επαλήθευση υλικού.

Η ανάλυση δομών αποτελείται από αρκετές συγκεκριμένες τεχνικές όπως η σύνθεση, η χρονική ανάλυση και ο έλεγχος ισότητας.

Η προσομοίωση είναι ένα είδος testing. Ένα ξαναδιαμορφώσιμο γενικό σύστημα υλικού διαμορφώνεται έτσι ώστε να συμπεριφέρεται σαν το κύκλωμα υπό εξέταση και εξετέζεται ενδελεχώς. Όπως και με το testing λογισμικού, η προσομοίωση παρέχει ένα σύνολο από ερεθίσματα στο κύκλωμα και συγκρίνει τις γενικευμένες εξόδους με τις αναμενόμενες εξόδους όπως περιγράφηκαν στις προδιαγραφές του chip. Για να ελεγχθεί πλήρως το κύκλωμα, όλοι οι πιθανοί συνδυασμοί σε όλες τις πιθανές καταστάσεις του συστήματος πρέπει να εξεταστούν. Αυτό δεν είναι πρακτικό και ο αριθμός των δοκιμών πρέπει να μειωθεί σημαντικά, αυξάνοντας όμως την πιθανότητα να μην βρεθούν κάποια λάθη.

Με την εξομοίωση, ένα μοντέλο του συστήματος κατασκευάζεται και εξομοιώνεται. Τα μοντέλα συνήθως παραχωρούνται χρησιμοποιώντας περιγραφικές γλώσσες όπως η Verilog και η VHDL. Βάση των ερεθισμάτων, τα μονοπάτια εκτέλεσης του μοντέλου εξετάζονται χρησιμοποιώντας έναν εξομοιωτή. Αυτά τα ερεθίσματα παρέχονται από κάποιον χρήστη ή από κάποια αυτοματοποιημένα μέσα όπως μία τυχαία γεννήτρια. Ένας κακός συνδυασμός μεταξύ της εξόδου του εξομοιωτή και της εξόδου που περιγράφεται από τις προδιαγραφές δείχνει την ύπαρξη λάθους. Ο εξομοιωτής είναι όπως το testing αλλά εφαρμόζεται σε μοντέλα. Έχει όμως το ίδιο μειονέκτημα με τον προσομοιωτή. Ο αριθμός των σεναρίων που πρέπει να ελεγχθούν σε ένα μοντέλο είναι πολύ περισσότερα από αυτά που ελέγχονται στην πραγματικότητα.

Η εξομοίωση είναι η πιο γνωστή τεχνική για επαλήθευση υλικού και χρησιμοποιείται σε διάφορα στάδια σχεδιασμού. Εκτός από αυτές τις τεχνικές για ανίχνευση λαθών, χρειάζεται και testing υλικού για να βρεθούν λάθη επεξεργασίας.

2.6 Πλεονεκτήματα

- Είναι μία γενική μέθοδος επαλήθευσης η οποία είναι εφαρμόσιμη σε ένα μεγάλο πεδίο εφαρμογών όπως είναι τα ενσωματωμένα συστήματα (embedded systems), η τεχνολογία λογισμικού και ο σχεδιασμός υλικού (hardware design).

- Υποστηρίζει μερική επαλήθευση, δηλαδή για παράδειγμα οι ιδιότητες μπορούν να ελεγχθούν ξεχωριστά, έτσι ώστε να ελεγχθούν οι βασικές ιδιότητες πρώτα. Δεν χρειάζεται πλήρης προδιαγραφή των απαιτήσεων.
- Δεν είναι ευπαθές στην πιθανότητα ότι ένα λάθος είναι εκτεθειμένο. Αυτό αντιτίθεται με τον έλεγχο και την προσομοίωση που στοχεύουν στο να ανιχνεύσουν τις πιο πιθανές ατέλειες.

- Παραχωρεί διαγνωστικές πληροφορίες σε περίπτωση που μια ιδιότητα ανατραπεί. Αυτό είναι πολύ χρήσιμο για σκοπούς αποσφαλμάτωσης.

- Είναι μια πιθανή «push-button» τεχνολογία. Η χρήση του model-checking δεν απαιτεί σε μεγάλο βαθμό ούτε την αλληλεπίδραση του χρήστη ούτε εμπειρία.

- Προκαλεί ένα γοργά αυξανόμενο ενδιαφέρον από τη βιομηχανία. Πολλές εταιρείες υλικού έχουν αρχίσει εργαστήρια επαλήθευσης, εργασίες με απαιτούμενη δεξιότητα για model-checking εμφανίζονται συχνά και εμπορικοί ελεγκτές μοντέλων έχουν γίνει διαθέσιμοι.

- Μπορεί εύκολα να ενσωματωθεί σε ήδη υπάρχοντες κύκλους ανάπτυξης. Μελέτες έχουν δείξει ότι μπορεί να οδηγήσει σε χαμηλότερο χρόνο ανάπτυξης.

- Έχει μαθηματικό υπόβαθρο. Βασίζεται στη θεωρία αλγόριθμων γραφικών παραστάσεων, δομές δεδομένων και λογικής.

2.7 Μειονεκτήματα

- Είναι κυρίως κατάλληλο για control-intensive εφαρμογές και λιγότερο για data-intensive εφαρμογές αφού τα δεδομένα συνήθως κυμαίνονται πάνω από άπειρες περιοχές.

- Η δυνατότητα εφαρμογής του υπόκειται σε θέματα αποφασιστικότητας. Για συστήματα άπειρων καταστάσεων το model-checking είναι γενικά μη αποτελεσματικά υπολογίσιμο.

- Επαληθεύει ένα μοντέλο του συστήματος και όχι το πραγματικό σύστημα. Κάθε παραγόμενο αποτέλεσμα είναι όσο καλό όσο το μοντέλο του συστήματος. Επιπλέον τεχνικές όπως είναι οι δοκιμές (testing) χρειάζονται για να βρεθούν ελαττώματα επεξεργασίας (για το hardware) ή λάθη κώδικα (για το software).

- Ελέγχει μόνο δηλωμένες απαιτήσεις άρα δεν υπάρχει εγγύηση για πληρότητα. Η ισχύς των ιδιοτήτων που δεν ελέγχονται δεν μπορούν να κριθεί.

- Υπάρχει το πρόβλημα του state-space explosion, δηλαδή ο αριθμός των καταστάσεων που που χρειάζονται για να μοντελοποιηθεί το σύστημα με ακρίβεια μπορούν εύκολα να υπερβαίνουν το ποσοστό της ελεύθερης μνήμης του υπολογιστή. Παρόλη την ανάπτυξη πολλών αποτελεσματικών μεθόδων για να ξεπεραστεί αυτό το πρόβλημα, τα μοντέλα ρεαλιστικών συστημάτων μπορεί να είναι ακόμα πολύ μεγάλα για να χωρέσουν στη μνήμη.

- Η χρήση του απαιτεί κάποια εμπειρία στο να βρεθούν κατάλληλες αφαιρέσεις ώστε να ληφθούν μικρότερα μοντέλα και στο να δηλωθούν οι ιδιότητες στο λογικό φορμαλισμό που χρησιμοποιείται.

- Δεν επιτρέπει τον έλεγχο γενικεύσεων. Τα συστήματα ελέγχου με αυθαίρετο αριθμό στοιχείων δεν μπορούν να αντιμετωπιστούν. Το model-checking μπορεί όμως να

προτείνει αποτελέσματα για αυθαίρετες παράμετρους που μπορούν να επαληθευθούν χρησιμοποιώντας βοηθούς αποδείξεως.

Κεφάλαιο 3

Τεχνικές μοντελοποίησης

3.1 Πίνακες απόφασης

Οι πίνακες απόφασης είναι ένας ακριβής αλλά και ταυτοχρόνως συμπαγής τρόπος για μοντελοποίηση περίπλοκης λογικής. Συσχετίζει συνθήκες με ενέργειες, όπως για παράδειγμα τις εντολές if-else.

Αποτελείται από τέσσερα τεταρτημόρια όπως φαίνεται και στον πίνακα πιο κάτω :

Συνθήκες	Εναλλακτικές Συνθήκες
Ενέργειες	Είσοδοι ενεργειών

Πίνακας 3.1 : Απεικόνιση του πίνακα απόφασης

Κάθε απόφαση αντιστοιχεί σε μια μεταβλητή, μια σχέση ή ένα κατηγόρημα των οποίων οι πιθανές τιμές παρατίθενται μεταξύ των εναλλακτικών συνθηκών. Κάθε ενέργεια είναι μια διαδικασία ή λειτουργία που θα εκτελεστεί και οι είσοδοι καθορίζουν αν ή/και με ποια σειρά η ενέργεια θα εκτελεστεί για το σύνολο των εναλλακτικών συνθηκών που αντιστοιχεί η είσοδος. Πολλές φορές συναντούμε το σύμβολο ‘-‘ το οποίο σημαίνει «δεν μας ενδιαφέρει». Χρησιμοποιώντας αυτό το σύμβολο απλοποιούμε τους πίνακες απόφασης, ειδικά σε περιπτώσεις όπου η σημασία της συνθήκης έχει μικρή επίρροια στις ενέργειες που θα εκτελεστούν.

Εκτός από τη βασική δομή τεσσάρων τεταρτημόριων, οι πίνακες απόφασης ποικίλλουν ευρέως ανάλογα με τον τρόπο που οι εναλλακτικές συνθήκες και οι είσοδοι ενεργειών αντιπροσωπεύονται. Μερικοί πίνακες απόφασης χρησιμοποιούν τις απλές true/false τιμές για να αντιπροσωπεύσουν τις εναλλακτικές συνθήκες, ενώ άλλοι πίνακες μπορεί να χρησιμοποιούν τις αριθμημένες εναλλακτικές περιπτώσεις και μερικοί πίνακες χρησιμοποιούν ακόμη και τη συγκεχυμένη λογική (fuzzy logic) ή τις πιθανολογικές αντιπροσωπεύσεις (probabilistic representation) για τις εναλλακτικές συνθήκες. Με παρόμοιο τρόπο, οι είσοδοι ενεργειών μπορούν απλά να αντιπροσωπεύουν αν μια δράση πρόκειται να εκτελεστεί, ή στους πιο προηγμένους πίνακες απόφασης, την αλληλουχία των ενεργειών που εκτελούν.

Οι πίνακες απόφασης μπορούν επίσης να είναι ενσωματωμένοι σε προγράμματα του υπολογιστή και να χρησιμοποιηθούν για να οδηγήσουν τη λογική του προγράμματος. Για παράδειγμα, μπορεί να είναι ένας συμβουλευτικός πίνακας που θα περιέχει ένα πεδίο πιθανών τιμών εισόδου και έναν δείκτη λειτουργίας στο σημείο του κώδικα που επεξεργάζεται αυτή την είσοδο. Αυτή η μέθοδος είναι στατική.

Είσοδος	Δείκτης Λειτουργίας
‘1’	Λειτουργία 1 (αρχικοποίηση)
‘2’	Λειτουργία 2 (επεξεργασία του ‘2’)
‘9’	Λειτουργία 9 (τερματισμός)

Πίνακας 3.2 : Απεικόνιση του ενσωματωμένου πίνακα απόφασης σε πρόγραμμα

Χρήση πίνακα απόφασης για model-checking :

Θεωρούμε ότι έχουμε ένα απλό σύστημα το οποίο θέλουμε να ελέγξουμε με ένα μοντέλο. Θα δημιουργήσουμε έναν πίνακα απόφασης στον οποίο θα βάλουμε στην στήλη των εισόδων τις εισόδους που θα βάλουμε στο σύστημα και στη στήλη του δείκτη λειτουργίας τις λειτουργίες που πρέπει να τεθούν σε εφαρμογή ανάλογα με την είσοδο που έχουμε θέσει στο σύστημα. Τις λειτουργίες τις βρίσκουμε από τις προδιαγραφές του συστήματος. Στη συνέχεια, θα βάλουμε τις εισόδους στο σύστημα

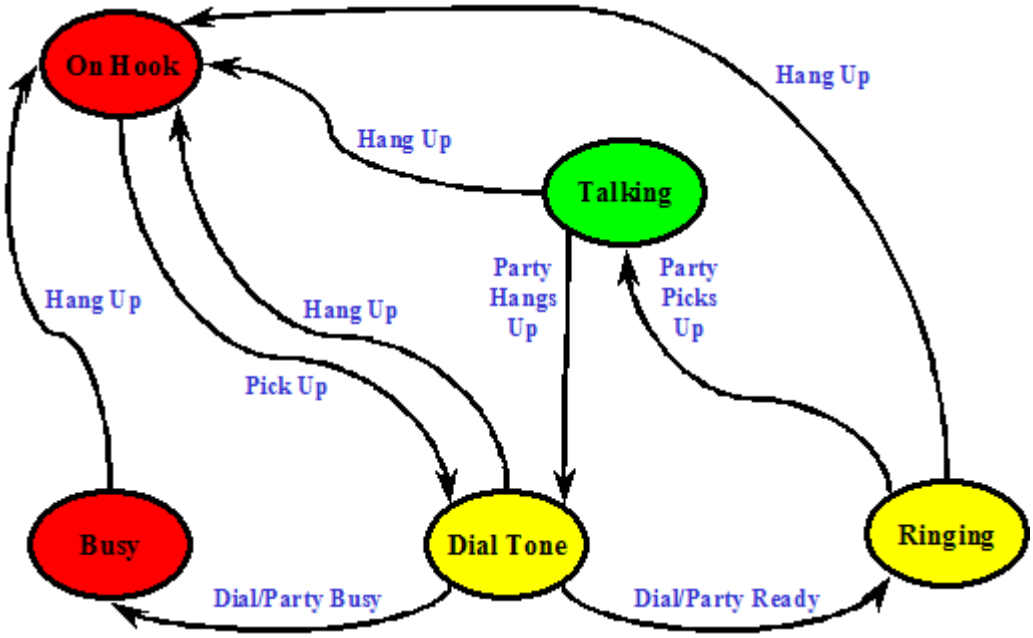
και θα συγκρίνουμε τις λειτουργίες που τίθονται σε εφαρμογή με αυτές του πίνακα (δηλαδή αυτές που έπρεπε να εκτελεστούν).

3.2 Μηχανές πεπερασμένων καταστάσεων (finite state machines)

Μία μηχανή πεπερασμένων καταστάσεων (αλλιώς ονομάζεται και αυτόματο πεπερασμένων καταστάσεων) είναι ένα μοντέλο συμπεριφοράς που αποτελείται από πεπερασμένο αριθμό καταστάσεων, μεταβάσεις μεταξύ των καταστάσεων αυτών και ενέργειες.
Αντιπροσωπεύεται ως (I, S, T, F, L) όπου :
- I είναι το σύνολο των εισόδων στο σύστημα
- S είναι το σύνολο όλων των καταστάσεων του συστήματος
- T είναι μια συνάρτηση που καθορίζει αν μια μετάβαση προκύπτει όταν μια είσοδος εφαρμόζεται στο σύστημα σε μια συγκεκριμένη κατάσταση
- F είναι το σύνολο των τελικών καταστάσεων
- L είναι το σύνολο των αρχικών καταστάσεων

Πώς χρησιμοποιείται για model-checking :

Ας υποθέσουμε ότι έχουμε ένα απλό τηλεφωνικό σύστημα. Οι κόμβοι είναι οι καταστάσεις του τηλεφώνου. Οι ακμές αντιπροσωπεύουν ενέργειες που μπορεί ο χρήστης να εκτελέσει (αυτές είναι και οι είσοδοι του συστήματος). Οι περιπτώσεις δοκιμής καθορίζουν μια σειρά εισόδων, τις καταστάσεις που το σύστημα αναμένεται να βρίσκεται μετά από κάθε ενέργεια και τις τιμές όλων των εξόδων του συστήματος.



Σχήμα 3.1 : Απεικόνιση μηχανής πεπερασμένων καταστάσεων για τηλέφωνο [4]

Το I είναι {Dial/Party Busy, Dial/Party Ready, Hang Up, Party Hangs Up, Party Picks Up, Pick Up}.
Το S είναι {Busy, Dial Tone, On-Hook, Ringing, Talking}.
Το L είναι {On-Hook}.
Το F είναι {On-Hook}.

Αυτή η μηχανή πεπερασμένων καταστάσεων μπορεί να χρησιμοποιηθεί για να παραχθούν περιπτώσεις δοκιμής.

Action	State	Action	State
	On-Hook	Dial/Party Ready	Ringing
Pick Up	Dial Tone	Party Picks Up	Talking
Dial/Party Busy	Busy	Party Hangs Up	Dial Tone
Hang Up	On-Hook	Hang Up	On-Hook
Pick Up	Dial Tone	Pick Up	Dial Tone
Dial/Party Ready	Ringing	Dial/Party Ready	Ringing
Hang Up	On-Hook	Party Picks Up	Talking
Pick Up	Dial Tone	Hang Up	On Hook

Πίνακας 3.3 : Περιπτώσεις δοκιμής [4]

Ο πιο πάνω πίνακας δείχνει μια ακολουθία από 5 εισόδους. Αυτή η ακολουθία αρχίζει και τελειώνει με την κατάσταση ‘On-Hook’. Έχει την επιθυμητή ιδιότητα του να εκτελέσει κάθε πιθανή ενέργεια σε κάθε κατάσταση τουλάχιστον μία φορά. Αυτή η ιδιότητα αποκαλείται κάλυψη ενέργειας (action coverage). Η ακολουθία όμως δεν είναι μοναδική. Χρησιμοποιώντας το μοντέλο για να παράγουμε ακολουθίες δοκιμών, μπορεί να έχει ως αποτέλεσμα πολλές ακολουθίες δοκιμών, εκ των οποίων η κάθε μία να έχει κάτι διαφορετικό αλλά να καλύπτει το ίδιο κριτήριο κάλυψης.

Για να λύσουμε αυτό το πρόβλημα θα μπορούσαμε να αρχίζουμε άλλη περίπτωση δοκιμής κάθε φορά που το σύστημα μπαίνει στην κατάσταση {On-Hook}. Τέσσερις τέτοιες περιπτώσεις φαίνονται στον πιο πάνω πίνακα (κάθε περίπτωση αρχίζει με το On-Hook της προηγούμενης και τελειώνει στο πρώτο εμφανιζόμενο On-Hook). Άλλες ακολουθίες δοκιμής που καλύπτουν όλες τις ενέργειες, θα είχαν διαφορετικό αριθμό περιπτώσεων δοκιμής. Μια περίπτωση δοκιμής θα πρέπει να καθορίζει τις αναμενόμενες αξόδους καθώς επίσης και τις ακολουθίες των εισόδων.

Η κάλυψη ενέργειας είναι εύκολο να επιτευχθεί. Κάτι που είναι πιο δύσκολο, όμως είναι και πιο αποτελεσματικό, είναι η μεταστροφή κάλυψης (switch coverage). Η μεταστροφή κάλυψης επιτυγχάνεται όταν για κάθε κατάσταση, κάθε ζεύγος ενεργειών που οδηγεί είτε εντός είτε εκτός της κατάστασης καλύπτεται στην ακολουθία δοκιμής. Για παράδειγμα ας δούμε την κατάσταση ‘Dial-Tone’. Ένα πιθανό μονοπάτι που οδηγεί σε αυτή την κατάσταση είναι η ακολουθία (‘Party Hangs Up’, ‘Dial Party Ready’). Ο πιο πάνω πίνακας δεν καλύπτει αυτή την περίπτωση.

Ο πιο κάτω πίνακας δείχνει μια ακολουθία δοκιμής που πετυχαίνει μεταστροφή κάλυψης. Παρατηρούμε όμως ότι είναι μεγαλύτερος από τον πιο πάνω πίνακα. Αυτό συμβαίνει όταν θέλουμε να πετύχουμε μεταστροφή κάλυψης αντί για κάλυψη ενέργειας.

Action	State	Action	State
	On-Hook	Dial/Party Ready	Ringing
Pick Up	Dial Tone	Party Picks Up	Talking
Hang Up	On-Hook	Party Hangs Up	Dial Tone
Pick Up	Dial Tone	Hang Up	On-Hook
Dial/Party Busy	Busy	Pick Up	Dial Tone
Hang Up	On-Hook	Dial/Party Ready	Ringing
Pick Up	Dial Tone	Party Picks Up	Talking
Dial/Party Ready	Ringing	Party Hangs Up	Dial Tone
Hang Up	On-Hook	Dial/Party Ready	Ringing
Pick Up	Dial Tone	Party Picks Up	Talking
Dial/Party Ready	Ringing	Party Hangs Up	Dial Tone
Party Picks Up	Talking	Dial/Party Busy	Busy
Hang Up	On-Hook	Hang Up	On-Hook
Pick Up	Dial Tone		

Πίνακας 3.4 : Περιπτώσεις δοκιμής [4]

Στον πιο πάνω πίνακα παρατηρούμε ότι αν πάρουμε για παράδειγμα την αρχική κατάσταση ‘On-Hook’ για να δημιουργήσουμε περιπτώσεις δοκιμής, θα έχουμε έξι σε αντίθεση με τις τέσσερις που είχαμε πριν.

3.3 Γραμματικές (Grammars)

Μια γραμματική είναι ένα σύνολο από κανόνες σχηματισμού που περιγράφουν ποια strings που δημιουργούνται από το αλφάβητο μιας γλώσσας είναι συντακτικά σωστά μέσα στη γλώσσα. Δείχνει μόνο την τοποθεσία και το χειρισμό των strings της γλώσσας, δηλαδή δεν περιγράφει τίποτα άλλο για την γλώσσα όπως π.χ. τι σημαίνει το string. Συνήθως χρησιμοποιούνται για να κατασκευαστούν μεταγλωττιστές διαφόρων γλωσσών προγραμματισμού.

Παράδειγμα γραμματικής :

S ->aSb

Το aaabbb ανήκει στη γλώσσα ενώ το abaaabb δεν ανήκει.

Πώς χρησιμοποιούνται για model-checking :

Έχουμε για παράδειγμα τον πιο κάτω κώδικα :

```
static void TestCase(){
while( Fun("L21") ){
L22: break; }}
else {
L3:{
L31: gotoL22;
L32: gotoL31; }}
```

Το μοντέλο μας χρησιμοποιώντας γραμματικές θα είναι κάτι σαν αυτό :

Statement :: if (BooleanCondition) { Statement } else { Statement }

Statement :: while (BooleanCondition) { Statement }

Statement :: { Statement Statement}

Statement :: break;

Statement :: gotoLabel;

Statement :: Label : Statement

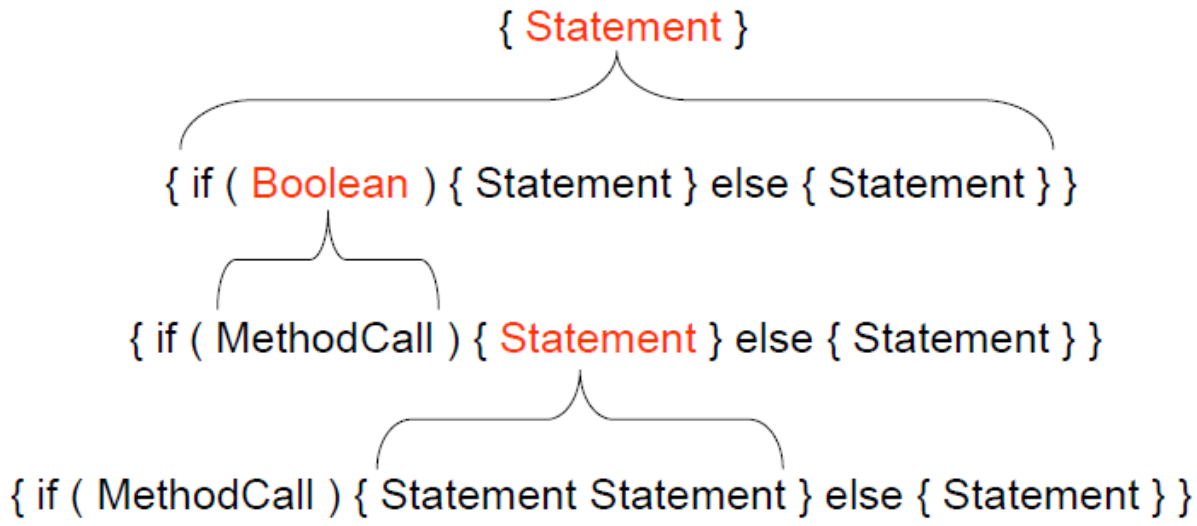
Statement :: ExpressionStatement;

ExpressionStatement:: MethodCall

BooleanExpression:: MethodCall

MethodCall:: Fun(Label)

Η γραφική αναπαράσταση αυτού του μοντέλου είναι η εξής :



Σχήμα 3.2 : Γραφική αναπαράσταση γραμματικής του μοντέλου [4]

Οι δοκιμές που παράγονται από τη γραμματική είναι οι εξής :

suite123.cs

```
static void TestCase(){
L:
if ( Fun("L1") ){
L2:
if ( Fun("L21") ){
L22: goto L;
}
else{
L23: goto L;
}}
else{
L3:
{
L31: goto L32;
L32: goto L;
}}}
```

suite321.cs

```
static void TestCase(){
L:
if ( Fun("L1") ){
L2:
while( Fun("L21") ){
L22:
break;
}}
else{
L3:
{
L31: goto L;
L32: goto L31;
}}}
```

Σχήμα 3.3 : Δοκιμές παραγόμενες από τη γραμματική [4]

3.4 Αλυσίδες Markov (Markov chains)

Μια αλυσίδα Markov είναι μια στοχαστική διαδικασία με την ιδιότητα Markov. Η ιδιότητα Markov σημαίνει ότι δεδομένης μιας παρούσας κατάστασης, οι μελλοντικές καταστάσεις είναι ανεξάρτητες από τις προηγούμενες καταστάσεις. Δηλαδή, η παρούσα κατάσταση περιέχει όλες τις πληροφορίες που μπορεί να επηρεάσουν την μελλοντική εξέλιξη της διαδικασίας. Το ότι είναι στοχαστική σημαίνει ότι όλες οι μεταβάσεις καταστάσεων είναι πιθανολογικές.

Σε κάθε βήμα, το σύστημα μπορεί να αλλάξει την κατάστασή του από την παρούσα κατάσταση σε άλλη κατάσταση σύμφωνα με μια πιθανολογική διανομή. Οι αλλαγές της κατάστασης λέγονται μεταβάσεις και οι πιθανότητες που σχετίζονται με διάφορες αλλαγές καταστάσεων λέγονται πιθανότητες μετάβασης.

Ένα παράδειγμα μιας αλυσίδας Markov είναι μια τυχαία διαδρομή στην αριθμητική γραμμή που αρχίζει από το 0 και μεταβαίνει +1 ή -1 με ίσα πιθανότητα σε κάθε βήμα.

Συμβολίζεται : $Pr(X_{n+1} = x \mid X_n = x_n, \dots, X_1 = x_1) = Pr(X_{n+1} = x \mid X_n = x_n)$
Οι θετικές τιμές του X αποτελούν ένα πεπερασμένο σύνολο που ονομάζεται χώρος καταστάσεων της αλυσίδας. Οι αλυσίδες συχνά περιγράφονται με έναν κατευθυνόμενο γράφο.

Πώς χρησιμοποιούνται για model-checking :

Οι αλυσίδες Markov μπορούν για ευκολία να θεωρηθούν σαν μηχανές πεπερασμένων καταστάσεων με βάρη τις πιθανότητες των μεταβάσεων. Η συλλογή των πιθανοτήτων δημιουργεί ένα profile. Για να κάνουμε model-checking με αλυσίδες Markov χρησιμοποιούμε μια τυχαία διαδικασία για να παραχθούν οι περιπτώσεις δοκιμής που προσαρμόζονται στο profile. Λέμε ότι έχουμε «usage model» και «usage profile» όταν οι πιθανότητες καθορίζουν τη χρήση του συστήματος. Για παράδειγμα, η προσέγγιση «Big Bang» της δοκιμής ολοκλήρωσης (integration testing) ορίζεται ως Usage Model Testing. Σε αυτή την περίπτωση, η ανάλυση των αποτελεσμάτων των περιπτώσεων δοκιμής παρέχει στο σύστημα αξιοπιστία δοκιμής.

Οι αλυσίδες Markov χρησιμοποιούνται για statistical model-checking, το οποίο είναι ισχυρό για τη βελτίωση της διαδικασίας των δοκιμών. Λόγω του ότι χρησιμοποιεί πιθανότητες, η τυχαία παραγωγή περιπτώσεων δοκιμής ελέγχεται και οδηγείται για να συγκλίνει στο στόχο. Για παράδειγμα, με τη χρήση ενός «usage profile», οι παραγόμενες περιπτώσεις δοκιμής θα ελέγξουν τις πιο χρησιμοποιημένες διαδικασίες πριν τις άλλες.

Κεφάλαιο 4

Ακολουθιακά Διαγράμματα

4.1 Τι είναι τα ακολουθιακά διαγράμματα

Ο κύριος σκοπός ενός sequence diagram είναι να καθορίσει ακολουθίες γεγονότων που καταλήγουν σε κάποιο επιθυμητό αποτέλεσμα. Εστιαζόμαστε περισσότερο όχι στα μηνύματα αλλά στη σειρά με την οποία προκύπτουν. Περιγράφει πώς οι ομάδες αντικειμένων συνεργάζονται στο να επιτύχουν κάποια συμπεριφορά του συστήματος. Αυτή η συνεργασία εφαρμόζεται ως σειρά μηνυμάτων μεταξύ των αντικειμένων. Τα διαγράμματα ακολουθίας δεν είναι χρήσιμα για τη συμπεριφορά μέσα σε ένα αντικείμενο. Καθέτως σε ένα sequence diagram (από πάνω προς τα κάτω) βλέπουμε τη σειρά με την οποία προκύπτουν τα μηνύματα και οριζοντίως (ανάλογα με τη φορά του βέλους) βλέπουμε τα στιγμιότυπα των αντικειμένων από και στα οποία στέλνονται τα μηνύματα.

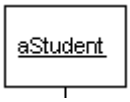
Επεξήγηση των μερών ενός sequence diagram :

Actor :

Αντιπροσωπεύει ένα εξωτερικό άτομο ή οντότητα που αλληλεπιδρά με το σύστημα

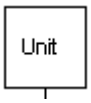
Object :

Αντιπροσωπεύει ένα αντικείμενο στο σύστημα ή ένα από τα στοιχεία του



Unit :

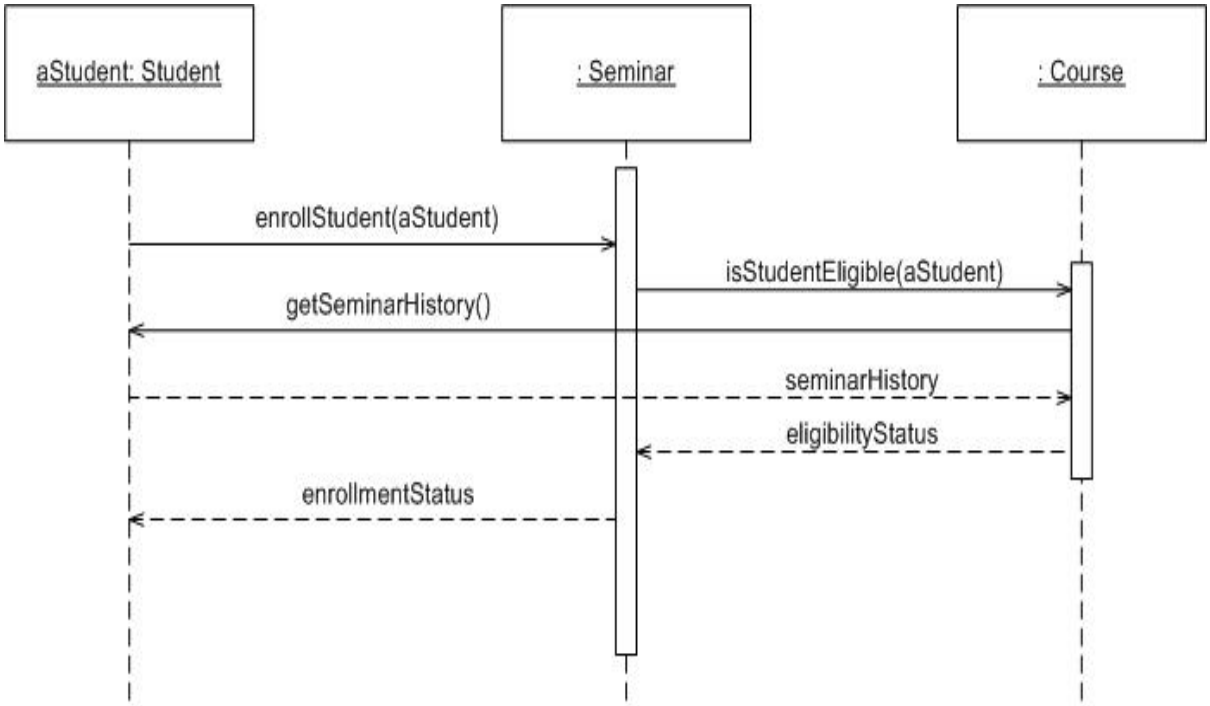
Αντιπροσωπεύει ένα υποσύστημα, στοιχείο, μονάδα ή άλλη λογική οντότητα στο σύστημα



Separator :

Αντιπροσωπεύει ένα όριο μεταξύ των υποσυστημάτων, των στοιχείων ή των μονάδων

Παράδειγμα :



Σχήμα 4.1 : Παράδειγμα ακολουθιακού διαγράμματος

4.2 Πώς χρησιμοποιούνται για model-checking

Για παράδειγμα έχουμε ένα πρόγραμμα και θέλουμε να το μοντελοποιήσουμε με sequence diagram για να το ελέγξουμε. Για να ελέγξουμε το πρόγραμμα πρέπει να ακολουθήσουμε κάποια βήματα.

Βήμα $1^{ο}$: ελέγχουμε αν υπάρχει η κατάλληλη μέθοδος στην κατάλληλη κλάση. Για το πιο πάνω παράδειγμα πρέπει να ελέγξουμε αν η μέθοδος enrollStudent() υπάρχει στην κλάση Seminar, αν η μέθοδος isStudentEligible() υπάρχει στην κλάση Course και ούτω καθεξής.

Βήμα $2^{ο}$: ελέγχουμε τη ροή με την οποία ανταλλάζονται τα μηνύματα. Αρχίζουμε από πάνω προς τα κάτω και ελέγχουμε αν εκτελούνται με τη σωστή σειρά.

Βήμα $3^{ο}$: ελέγχουμε αν επιστρέφονται οι σωστές πληροφορίες και στη σωστή χρονική στιγμή (αν χρειάζονται για να εκτελεστεί κάποια άλλη διεργασία).

Βήμα $4^{ο}$: ελέγχουμε αν πληρούνται οι προδιαγραφές. Για παράδειγμα αν έπρεπε να έπρεπε να κάνει όλες τις ενέργειες που κάνει, αν αφήνει κάποια πίσω κτλ.

Βήμα $5^{ο}$: ελέγχουμε αν τα αποτελέσματα που προκύπτουν με τις συγκεκριμένες εισόδους είναι εκείνα που έπρεπε να προκύπτουν.

Βήμα $6^{ο}$: κάνουμε βελτιστοποιήσεις στον κώδικα (π.χ. εντολές που δεν εκτελούνται, συνθήκες που μπορούμε να μειώσουμε την πολυπλοκότητα κτλ).

4.3 Μειονεκτήματα

Τα μειονεκτήματα των sequence diagrams για model-checking είναι τα εξής :

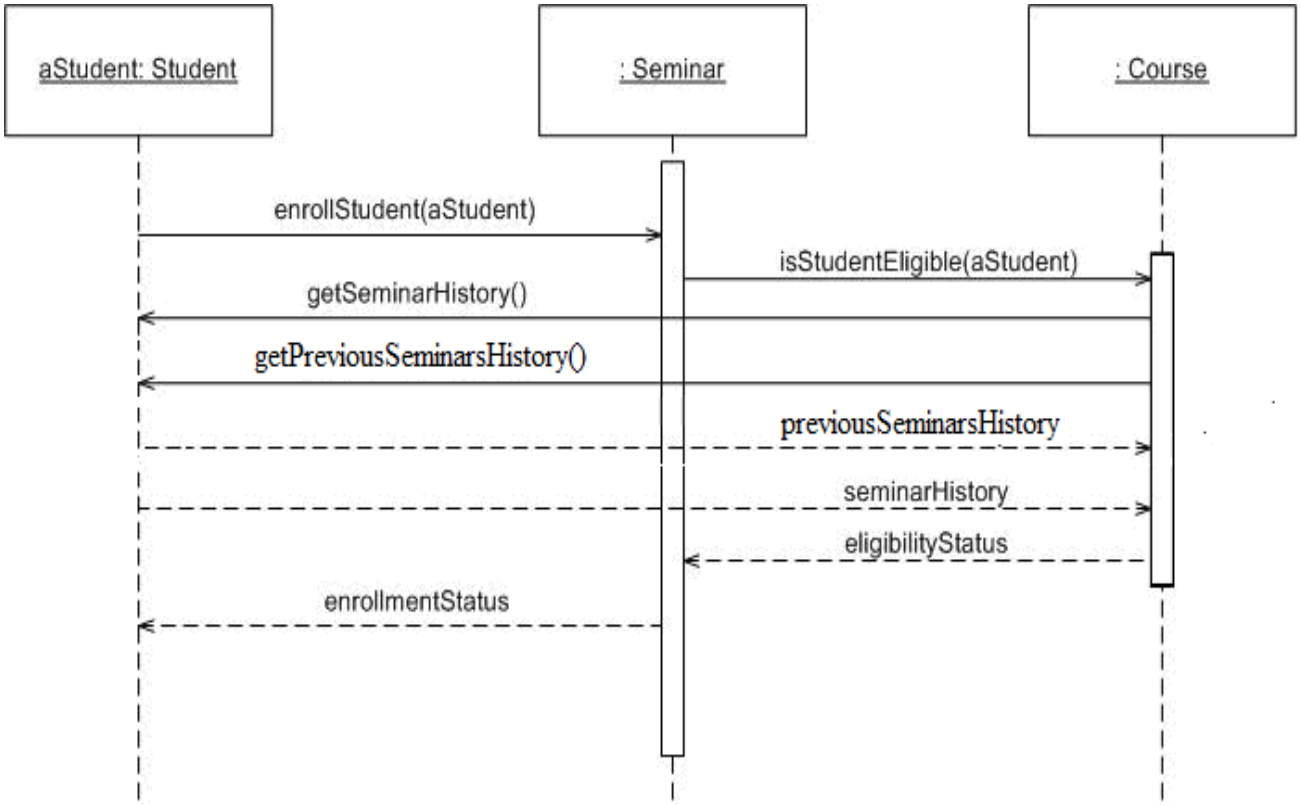
- Η αντιπροσώπευση περίπλοκων συστημάτων είναι δύσκολη
- Η αντιπροσώπευση παράλληλων ενεργειών δεν ικανοποιείται
- Η αντιπροσώπευση συνθηκών δεν ικανοποιείται
- Η αντιπροσώπευση ασύγχρονων μηνυμάτων δεν ικανοποιείται

4.4 Προτάσεις για λύσεις

Πιο κάτω θα δούμε τις προτάσεις που έκανα για βελτίωση του model-checking με ακολουθιακά διαγράμματα. Τα πιο κάτω διαγράμματα παρήχθηκαν από ένα πρόγραμμα της Java. Το πρόγραμμα παίρνει ως είσοδο κάποιο φοιτητή που θέλει να ενταχθεί σε κάποιο μάθημα, αλλά μπορεί να ενταχθεί υπό την προϋπόθεση ότι έχει παρακολουθήσει με επιτυχία ένα σεμινάριο πάνω σε εκείνο το μάθημα. Αν δεν έχει παρακολουθήσει το σεμινάριο εκείνο ή το παρακολούθησε αλλά όχι με επιτυχία (στο τέλος του σεμιναρίου είχαν γίνει ερωτήσεις για το θέμα), τότε πρέπει να δούμε αν έχει παρακολουθήσει κάποιο άλλο σεμινάριο με επιτυχία το οποίο να είναι σχετικό με το μάθημα εκείνο.

Όπως είπαμε πιο πάνω η αντιπροσώπευση παράλληλων ενεργειών δεν ικανοποιείται. Για παράδειγμα, θέλουμε να παίρνουμε ταυτόχρονα (με τη χρήση multi-threads για παράδειγμα στην Java) το αν παρακολούθησε ο φοιτητής το σεμινάριο με επιτυχία και το αν παρακολούθησε άλλα σεμινάρια σχετικά με το μάθημα με επιτυχία γιατί μπορεί ο καθηγητής να είχε περιορισμένο αριθμό θέσεων στο μάθημά του οπότε να ήθελε να διαλέξει τους φοιτητές που έχουν παρακολουθήσει τα περισσότερα σεμινάρια. Αυτό όμως, δηλαδή οι παράλληλες διεργασίες δεν φαίνονται στο διάγραμμα. Το sequence diagram θα δείχνει και τις δύο ενέργειες αλλά την μία κάτω από την άλλη, με αποτέλεσμα να μην γνωρίζουμε ότι εκτελούνται παράλληλα αν δεν γνωρίζουμε το σύστημα.

Έχουμε για παράδειγμα αυτό το sequence diagram :



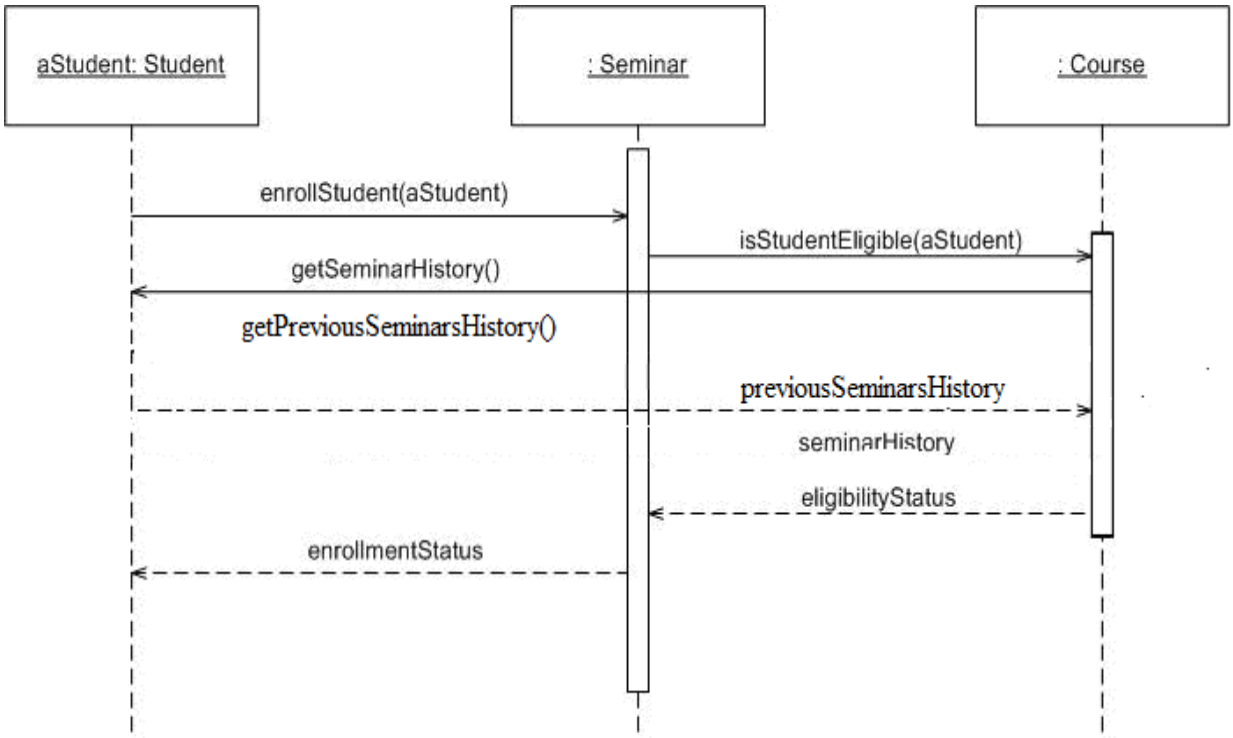
Σχήμα 4.2 : Ακολουθιακό διάγραμμα

Έχουμε δύο παράλληλες διεργασίες. Το getSeminarHistory() και το getPreviousSeminarsHistory(). Στο διάγραμμα φαίνονται και οι δύο διεργασίες αλλά δεν φαίνεται ότι εκτελούνται παράλληλα. Φαίνεται ότι εκτελείται η πρώτη και μετά η δεύτερη. Αυτό όμως στον έλεγχο του μοντέλου θα δημιουργήσει προβλήματα αφού δεν ελέγχεται σωστά, άρα και ο έλεγχος του συστήματος θα είναι λανθασμένος.

Για την επίλυση αυτού του προβλήματος προτείνω δύο λύσεις :

Πρώτη λύση :

Οι παράλληλες διεργασίες να φαίνονται στο ίδιο μήνυμα. Αυτό φαίνεται παρακάτω :

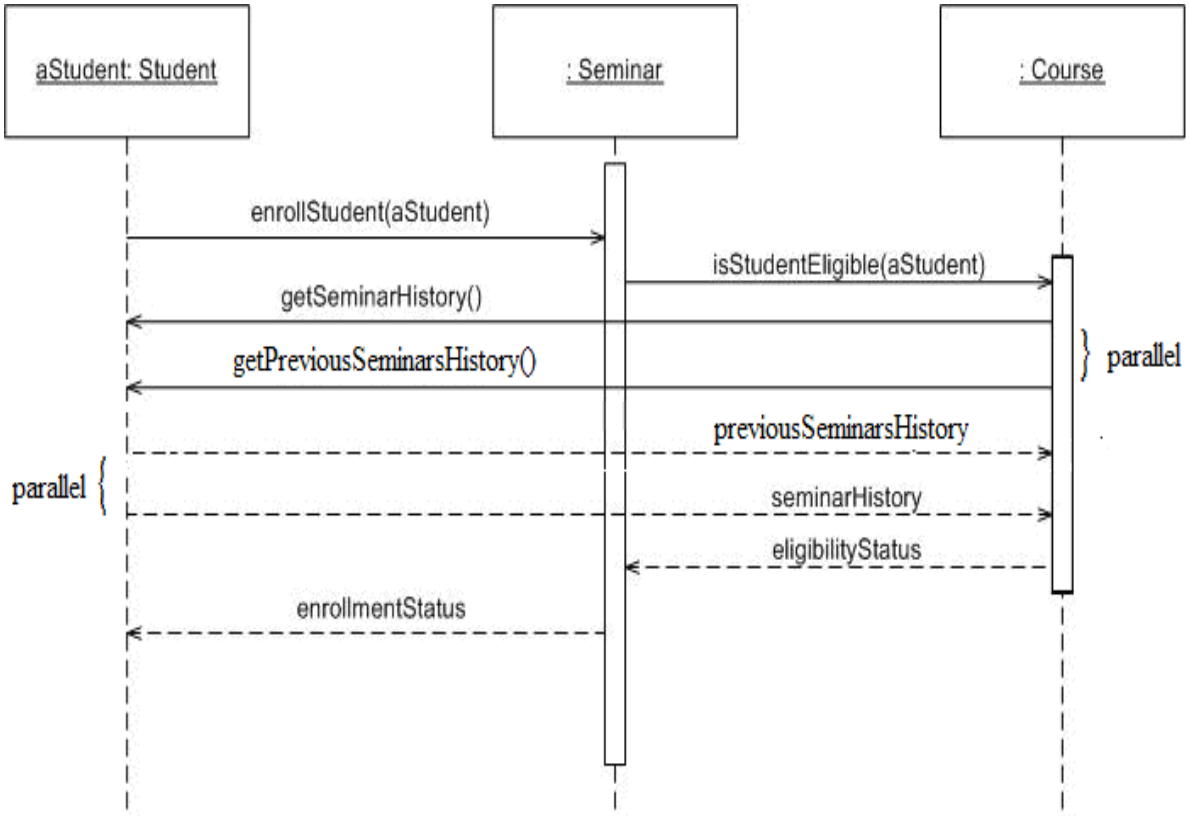


Σχήμα 4.3 : Πρώτη λύση ακολουθιακού διαγράμματος για παράλληλες διεργασίες

Βάζοντας τις δύο διεργασίες στο ίδιο μήνυμα φαίνεται ότι εκτελούνται την ίδια χρονική στιγμή. Έτσι, ο ελεγκτής του μοντέλου μπορεί να καταλάβει ότι οι μέθοδοι εκτελούνται ταυτόχρονα και τα αποτελέσματά τους επιστρέφονται ταυτόχρονα. Αυτή η λύση έχει ένα μειονέκτημα. Αν οι διεργασίες που εκτελούνται παράλληλα δεν πήγαιναν στην ίδια κλάση, στην περίπτωσή μας στην κλάση Student, τότε δεν θα μπορούσαμε να το απεικονίσουμε. Δηλαδή, αν η getSeminarHistory() ήταν στην κλάση Student και η getPreviousSeminarHistory() ήταν στην κλάση Seminar τότε το τόξο δεν θα μπορούσε να δείχνει και στις δύο. Άρα αυτή η λύση είναι προβληματική. Γι'αυτό είναι προτιμότερη η δεύτερη λύση που φαίνεται πιο κάτω.

Δεύτερη λύση :

Η δεύτερη λύση προκύπτει από κάτι που βρήκα ότι υποστηρίζει η UML για τη συνθήκη while. Βάζοντας το σύμβολο '{' εκεί που υπάρχει η ταυτοχρονία και γράφοντας την λέξη parallel δίπλα, καταλαβαίνουμε ότι οι διεργασίες εκτελούνται παράλληλα και ότι τα αποτελέσματά τους επιστρέφονται παράλληλα, δηλαδή την ίδια χρονική στιγμή.



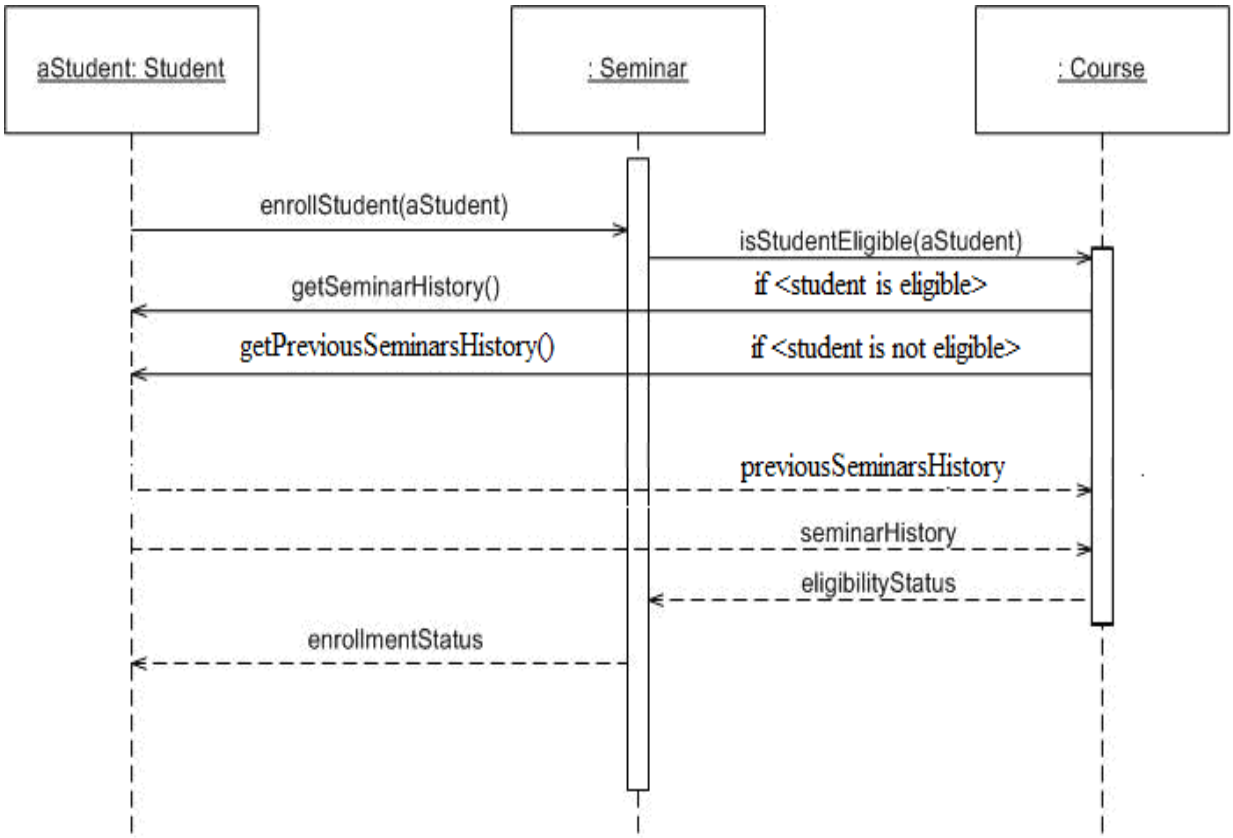
Σχήμα 4.4 : Δεύτερη λύση ακολουθιακού διαγράμματος για παράλληλες διεργασίες

Εδώ παρατηρούμε ότι δεν υπάρχει το πρόβλημα που υπήρχε πιο πάνω, δηλαδή αν οι δύο μέθοδοι βρίσκονταν σε διαφορετικές κλάσεις, θα μπορούσε και πάλι να απεικονιστεί με τον ίδιο τρόπο. Θα μπορούσαμε να έχουμε επίσης και περισσότερες από δύο ταυτόχρονες διεργασίες και πολύ απλά θα μεγάλωνε η αγκύλη ώστε να καλύπτει όλες τις παράλληλες διεργασίες.

Επίσης, όπως είπαμε πιο πάνω έχουμε και το πρόβλημα του ότι οι συνθήκη if-else δεν αντιπροσωπεύεται. Γι' αυτό το σκοπό προτείνω την πιο κάτω λύση.

Αν έχουμε για παράδειγμα :

```
if (student is eligible) {
  getSeminarHistory();
}
else{
  getPreviousSeminarsHistory();
}
```



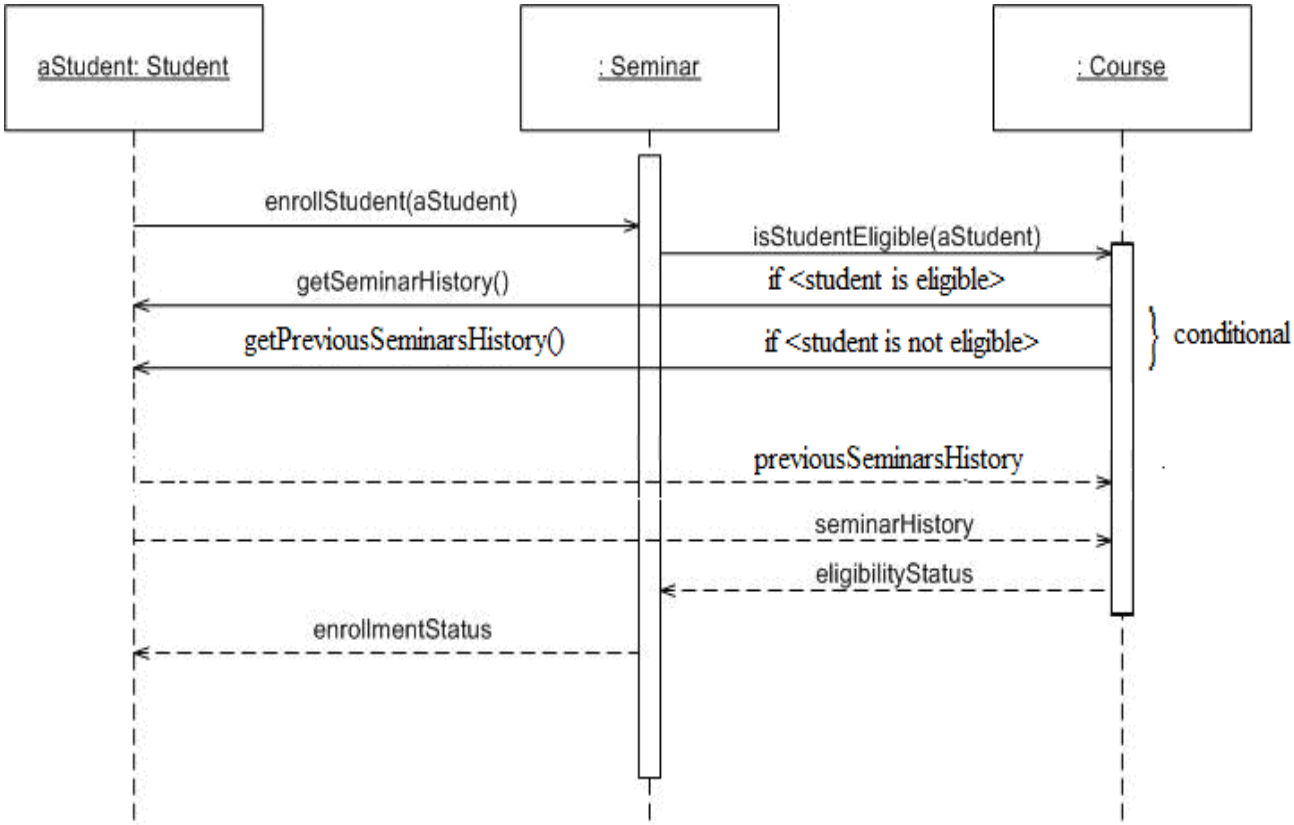
Σχήμα 4.5 : Λύση ακολουθιακού διαγράμματος για τη συνθήκη if-else

Αν για παράδειγμα ο καθηγητής του μαθήματος είχε σαν κριτήριο στο να αποδεχτεί στο μάθημά του κάποιον φοιτητή το αν παρακολούθησε το σεμινάριο με επιτυχία και αν όχι, ήθελε να δει ποια σεμινάρια παρακολούθησε με επιτυχία ώστε αν υπήρχε κάποιο

σχετικό με το μάθημα να τον επέλεγε τότε θα είχαμε τον πιο πάνω κώδικα. Και πάλι στο ακολουθιακό διάγραμμα θα φαίνονταν οι διεργασίες σαν να εκτελούνται η μία μετά την άλλη. Όμως αυτό θα αποτελούσε πρόβλημα γιατί δεν εκτελούνται και οι δύο διεργασίες αλλά εκτελείται είτε η μία είτε η άλλη. Και ανάλογα επιστρέφεται το κατάλληλο αποτέλεσμα. Αν φαίνονταν η μία μετά την άλλη τότε ο ελεγκτής του μοντέλου θα νόμιζε πως εκτελούνται και οι δύο διεργασίες και κατά συνέπεια θα έλεγχε λάθος το σύστημα, αφού η αναπαράστασή του είναι λανθασμένη.

Για να λυθεί αυτό το πρόβλημα μπορούμε απλά πάνω στο μήνυμα να βάλουμε τη συνθήκη. Στο δικό μας παράδειγμα αν βάλουμε τη συνθήκη “if student is eligible”, τότε αν ισχύει η πρόταση αυτή θα εκτελεστεί η πρώτη διεργασία δηλαδή η getSeminarHistory() και θα επιστραφεί το αποτέλεσμα της διεργασίας αυτής, δηλαδή το seminarHistory. Αν όμως δεν ισχύει η πρόταση αυτή, τότε δεν θα εκτελεστεί η πρώτη διεργασία αλλά μόνο η δεύτερη, δηλαδή η getPreviousSeminarHistory() και θα επιστραφεί το αποτέλεσμα αυτής της διεργασίας, δηλαδή το previousSeminarHistory().

Επίσης για να φαίνεται και πιο διακριτά θα μπορούσε το διάγραμμα να γίνει :



Σχήμα 4.6 : Βελτίωση λύσης ακολουθιακού διαγράμματος για τη συνθήκη if-else

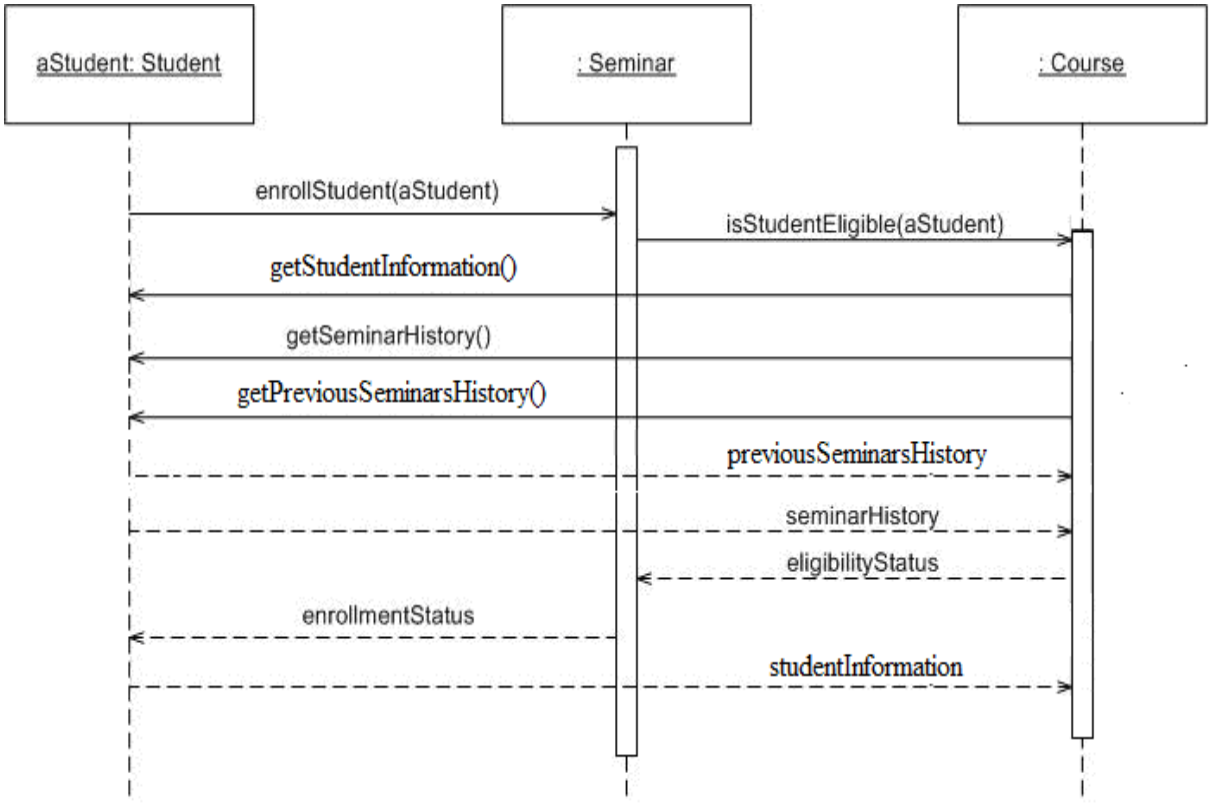
Χρησιμοποιούμε και πάλι το σύμβολο '}' για να δείξουμε ποιες διεργασίες βρίσκονται στην ίδια συνθήκη. Δίπλα από την αγκύλη βάζουμε τον όρο conditional για να δείξουμε ότι οι δύο διεργασίες υπόκεινται σε συνθήκη. Σε περίπτωση που είχαμε κι άλλες συνθήκες θα μπορούσαμε να βάλουμε κι άλλες αγκύλες. Σε άλλη περίπτωση που έχουμε τρεις προτάσεις σε μία συνθήκη (π.χ. if - else if - else), μπορούμε να βάλουμε και τις τρεις στην αγκύλη με την πρόταση της κάθε μίας στο μήνυμα.

Τέλος, όπως αναφέραμε δεν υποστηρίζονται τα ασύγχρονα μηνύματα (asynchronous messages). Ασύγχρονα μηνύματα έχουμε όταν για παράδειγμα στην Java εκτελεστεί μία μέθοδος και επειδή θέλει ώρα να εκτελεστεί μπορούν να εκτελούνται άλλες μέθοδοι μέχρι να επιστρέψει η πρώτη μέθοδος το απότελεσμα.

Στο δικό μας παράδειγμα, αν θέλαμε να πάρουμε τις πληροφορίες του φοιτητή αλλά θέλει ώρα για να επιστραφούν, για να μην περιμένουμε θα θέλαμε να εκτελεί άλλες διεργασίες που δεν χρειάζονται εκείνες τις πληροφορίες μέχρι να επιστραφεί το αποτέλεσμα.

Έχουμε δηλαδή την getStudentInformation() η οποία μπορεί να θέλει να ανατρέξει μια τεράστια βάση δεδομένων λόγω του ότι οι φοιτητές είναι πολλοί κι ως επακόλουθο, θα αργήσει να μας δώσει το αποτέλεσμα. Εμείς θέλουμε να εκτελεστούν οι επόμενες μέθοδοι και να μην χρειάζεται να περιμένουμε το αποτέλεσμα, γιατί έτσι ο χρόνος εκτέλεσης θα είναι μικρότερος και δεν θα χρειάζεται να περιμένει πολλή ώρα ο χρήστης. Οπότε, μέχρι να επιστραφεί το studentInformation (το αποτέλεσμα δηλαδή της getStudentInformation()) θέλουμε να εκτελέσει την getSeminarHistory() και την getPreviousSeminarHistory(). Αυτό όμως μπορεί να γίνει μόνο αν το αποτέλεσμα της getStudentInformation() δεν χρειάζεται για να εκτελεστούν οι άλλες δύο μέθοδοι. Αν χρειαζόταν, τότε θα έπρεπε να περιμένουμε να επιστραφεί το αποτέλεσμα πριν την εκτέλεση των δύο μεθόδων.

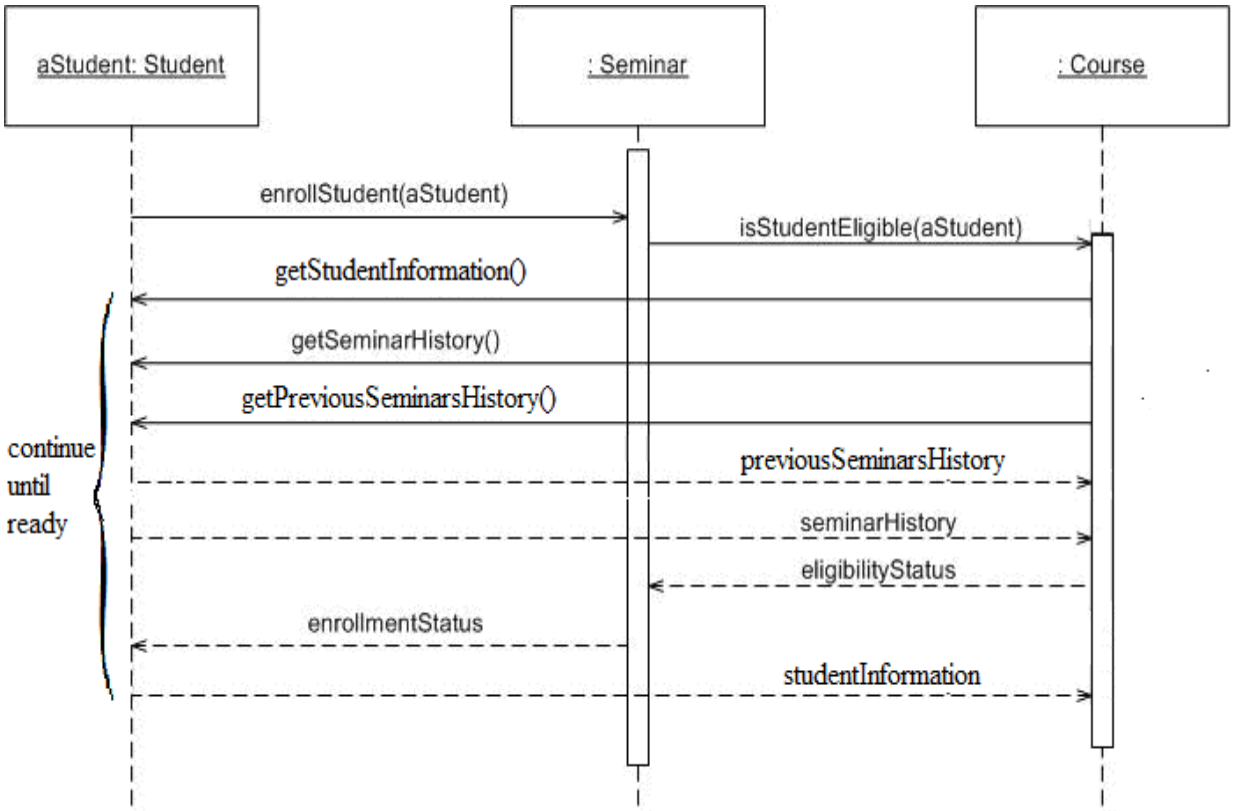
Η δική μου πρόταση είναι η εξής :



Σχήμα 4.7 : Λύση ακολουθιακού διαγράμματος για ασύγχρονα μηνύματα

Παρατηρούμε ότι το αποτέλεσμα της getStudentInformation(), δηλαδή το studentInformation, επιστρέφεται μετά την εκτέλεση της getSeminarHistory() και της getPreviousSeminarHistory() και μετά την επιστροφή των αποτελεσμάτων τους. Θα μπορούσε το αποτέλεσμα να επιστρέφεται και πιο πριν αν ήταν έτοιμο για να επιστραφεί. Όμως δεν φαίνεται διακριτά ότι το studentInformation είναι το αποτέλεσμα της getStudentInformation() κι έτσι μπορεί να ελεγχόταν λάθος το μοντέλο και κατά συνέπεια το σύστημα, αφού για παράδειγμα μπορεί να χρειάζονταν οι πληροφορίες του πρώτου αποτελέσματος για να εκτελεστεί η getPreviousSeminarHistory(). Γι'αυτό το λόγο προτείνω τη δεύτερη λύση η οποία είναι επέκταση της πρώτης.

Η δεύτερη λύση είναι η εξής :



Σχήμα 4.8 : Βελτίωση λύσης ακολουθιακού διαγράμματος για ασύγχρονα μηνύματα

Βάζοντας την αγκύλη και το χαρακτηρισμό “continue until ready”, καταλαβαίνουμε δύο περισσότερα πράγματα από την πιο πάνω λύση. Πρώτο, ότι έχουμε ασύγχρονα μηνύματα άρα δεν περιμένουμε να μας επιστραφεί ένα αποτέλεσμα πριν να εκτελεστεί η επόμενη μέθοδος και δεύτερο, ότι το studentInformation είναι το αποτέλεσμα της getStudentInformation() το οποίο επιστρέφεται όταν είναι έτοιμο. Άρα αυτή η λύση είναι καλύτερη από την πρώτη αφού μας δίνει περισσότερες και πιο σαφείς πληροφορίες για την ανταλλαγή των μηνυμάτων και ως συνέπεια ελέγχεται καλύτερα το μοντέλο και αυτό που μας ενδιαφέρει, το σύστημα.

Κεφάλαιο 5

Συμπεράσματα

5.1 Συμπεράσματα

Όπως καταλάβαμε από τις πιο πάνω πληροφορίες, το model-checking είναι μια πολύ χρήσιμη τεχνική στην ανάπτυξη λογισμικού. Τα λάθη πρέπει να διορθώνονται όσο το δυνατό πιο γρήγορα γιατί όσο πιο αργά τόσο περισσότερο το κόστος για να διορθωθούν. Το μοντέλο είναι ένα στιγμιότυπο του συστήματος, γι' αυτό ο έλεγχος είναι όσο καλός όσο το μοντέλο. Αν για παράδειγμα αναπαραστήσαμε λάθος το μοντέλο τότε τα αποτελέσματα του ελέγχου θα είναι λανθασμένα και ο έλεγχος δεν θα είναι σωστός. Το model-checking έχει κάποια μειονεκτήματα, όμως σε σύγκριση με το τι προσφέρει είναι ασήμαντα. Λόγω της χρησιμότητάς του, η βιομηχανία έχει δείξει μεγάλο ενδιαφέρον και έχει ως επακόλουθο να είναι ένας αναπτυσσόμενος τομέας άρα θα βρεθούν τρόποι για να εξαλειφθούν τα μειονεκτήματα αυτά. Για να επιτευχθεί το model-checking υπάρχουν πολλές τεχνικές, πολλές από αυτές όμως είναι ακόμα υπό ανάπτυξη.

Ο έλεγχος με τη χρήση ακολουθιακών διαγραμμάτων είναι αρκετά αντιπροσωπευτικός, όμως πρέπει να αναπτυχθούν περισσότερο ώστε να τον υποστηρίζουν πλήρως. Βρήκαμε ότι δεν υποστηρίζουν τις συνθήκες if-else, τις παράλληλες διεργασίες και τα ασύγχρονα μηνύματα.

Το να ελέγξουμε μόνο το ακολουθιακό διάγραμμα δεν είναι αρκετό. Πρέπει να βρούμε τρόπο να ελέγξουμε και τον κώδικα για να γίνουν βελτιώσεις ώστε να μειωθεί και ο χρόνος εκτέλεσης.

5.2 Μελλοντική δουλειά

Όσο αφορά τα ακολουθιακά διαγράμματα για model-checking θα μπορούσαμε στη συνέχεια να βρούμε και άλλα πράγματα που υπολείπονται από αυτά και να προτείνουμε και άλλες λύσεις. Ένα καλό βήμα θα ήταν να καταφέρουμε να ενσωματώσουμε αυτές τις λύσεις στα ακολουθιακά διαγράμματα. Έπειτα, μπορούμε να συνδιάσουμε τα ακολουθιακά διαγράμματα με άλλες μέθοδους μοντελοποίησης ώστε να καλύπτει η μία μέθοδος την άλλη για να πετύχουμε καλύτερο έλεγχο.

Όσο αφορά το model-checking θα μπορούσαμε να εξετάσουμε κι άλλες μέθοδους για να το πετύχουμε και να τις συγκρίνουμε. Επίσης, με το να εξετάσουμε και άλλες μεθόδους μπορούμε να βρούμε ποια μέθοδος αρμόζει σε κάθε περίπτωση καλύτερα από τις άλλες. Θα ήταν καλό αν συνδιάζαμε διάφορες μεθόδους μεταξύ τους, έτσι ώστε να συμπληρώνει η μία την άλλη και να μοντελοποιείται το σύστημα όσο καλύτερα γίνεται ώστε να ελέγχεται και το μοντέλο και κατά συνέπεια το σύστημα καλύτερα.

Βιβλιογραφία

[1] Christel Baier and Joost-Pieter Katoen : “Principles of model-checking”

[2] Anna Filippou : Σημειώσεις μαθήματος ΕΠΛ664

[3] Gerard J. Holzmann : “Software Analysis and Model Checking”

[4] Harry Robinson : “Model-based testing”

[5] James A. Whittaker and Ibrahim K. El-Far : “Model-based software testing”