Ατομική Διπλωματική Εργασία

ΜΟΝΤΕΛΟ-ΕΛΕΓΧΟΣ ΑΛΓΟΡΙΘΜΟΥ ΣΥΝΟΧΗΣ ΜΝΗΜΗΣ ΓΙΑ ΠΟΛΥΠΥΡΗΝΟΥΣ ΕΠΕΞΕΡΓΑΣΤΕΣ ΜΕ ΤΟ ΕΡΓΑΛΕΙΟ SPIN

Χριστοδουλίδης Ανδρέας Χρίστος

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2010

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μοντελο-έλεγχος με το εργαλείο SPIN

Χριστοδουλίδης Ανδρέας Χρίστος

Επιβλέπουσα Καθηγήτρια

Φιλίππου Άννα

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

Μάιος 2010

Ευχαριστίες

Θα ήθελα να ευχαριστήσω θερμά την επιβλέπουσα καθηγήτρια μου δρ. Άννα Φιλίππου για την συνεχή καθοδήγηση και για την υποστήριξη που μου έδινε κατά την διάρκεια της εκπόνησης αυτής της διπλωματικής εργασίας. Την ευχαριστώ για την υπομονή και την κατανόηση που έδειχνε στα διάφορα προβλήματα που αντιμετώπισα κατά καιρούς, και την βοήθεια που μου προσέφερε για την επίλυση των προβλημάτων αυτών. Η βοήθεια της και η συμβολή της έπαιξαν καταλυτικό ρόλο στην διεκπεραίωση της εργασίας αυτής.

Ευχαριστώ επίσης την οικογένεια μου για την υπομονή και την κατανόηση που επέδειξε, αλλά και για την συμπαράσταση και την αγάπη που μου έδειχνε στις δύσκολές στιγμές.

Περίληψη

Ο μοντελο-έλεγχος [1] είναι μια επαναστατική μέθοδος για την αυτοποιημένη επαλήθευση συστημάτων. Με την μέθοδο αυτή μπορούμε να ελέγξουμε κατα πόσο ένα σύστημα ικανοποιεί κάποιες τροπικές ιδιότητες, για παράδειγμα αν το σύστημα δίνει το σωστό αποτέλεσμα, αν περιέχει αδιέξοδο (deadlock), αν μπορεί να εμφανιστεί αδιέξοδο μέσα σε μια ώρα απο την εκκίνηση της εκτέλεσης του συστήματος, αν η απάντηση μπορεί να δωθεί μέσα σε κάποιο χρονικό διάστημα κλπ. Οι προγραμματιστές αφού κατανοήσουν σε βάθος τον τρόπο λειτουργίας του συστήματος, προσπαθούν να μοντελοποιήσουν σε κάποια γλωσσά επαλήθευσης μοντέλων τις βασικότερες λειτουργίες του, χρησιμοποιώντας κάποιο επίπεδο αφαιρετικότητας. Αφού μοντελοποιηθεί το σύστημα, χρησιμοποιείται κάποιο εργαλείο επαλήθευσης και ελέγχου μοντέλων, έτσι ώστε να εντοπιστούν λάθη αν υπάρχουν, και να διορθωθούν. Ο μοντελο-έλεγχος βασίζεται στον συστηματικό έλεγχο του συνόλου των καταστάσεων του συστήματος, και έχει την δυνατότητα να εντοπίσει λάθη, αλλά και να ελέγξει απαραίτητες ιδιότητες που πρέπει να πληροί το σύστημα.

Ένα εργαλείο που παρέχει δυνατότητες ελέγχου και επαλήθευσης μοντέλων είναι το SPIN [3], το οποίο χρησιμοποιεί την γλώσσα επαλήθευσης μοντέλων Promela [3] (Process Meta Language). Σκοπός αυτής της διπλωματικής εργασίας ήταν σε πρώτο στάδιο η γνωριμία και η εξοικείωση με το εργαλείο SPIN και της γλώσσας επαλήθευσης μοντέλων Promela που χρησιμοποιεί το SPIN για την υλοποίηση μοντέλων. Αυτό έγινε μέσω κάποιων ασκήσεων και μέσω υλοποίησης αλγορίθμων αμοιβαίου αποκλεισμού [3] και [4]. Στη συνέχεια έγινε μια μελέτη στο πεδίο των πολυπύρηνων συστημάτων, με έμφαση στην συνοχή της τοπικής μνήμης (cache) των πολυπύρηνων επεξεργαστών που είναι συνδεδεμένοι σε τοπολογία δακτυλίου. Ακολούθως επιλέχθηκε ένας αλγόριθμος από τον τομέα αυτό για ανάλυση και επαλήθευση με το συγκεκριμένο εργαλείο, ο οποίος υλοποιήθηκε στη γλώσσα Promela. Ο αλγόριθμος αυτός επαληθεύτηκε με το εργαλείο SPIN για εντοπισμό πιθανών αδιεξόδων (deadlocks), και αφού επαληθεύτηκε χωρίς λάθη, προχωρήσαμε στην επαλήθευση κάποιων ιδιοτήτων χρονικής λογικής που αφορούσαν την ορθότητα του αλγορίθμου. Τέλος έγινε κάποια αξιολόγηση των δυνατοτήτων του εργαλείου SPIN, βάσει της εμπειρίας που αποκομίσαμε από την χρήση του εργαλείου αυτού.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Στόχοι της Ατομικής Διπλωματικής Εργασίας

Τα τελευταία χρόνια έχουμε μαρτυρήσει μια καθολική επανάσταση στη χρήση συντρεχόντων και κατανεμημένων υπολογιστικών συστημάτων τα οποία εργάζονται και επικοινωνούν μεταξύ τους για να επιτελέσουν διάφορους στόχους υποστηρίζοντας τον σύγχρονο τρόπο ζωής. Εντούτοις, η πρόσφατη διείσδυση υπολογιστικών συστημάτων σε πολλούς τομείς της ζωής μας, και ιδιαίτερα η διαδεδομένη χρήση λογισμικού, συνοδεύτηκε από προβλήματα αξιοπιστίας, ασφάλειας και απόδοσης. Χαρακτηριστικά παραδείγματα σφαλμάτων που έχουν παρουσιαστεί είναι το διαστημόπλοιο Ariane-5 το οποίο εκτοξεύθηκε στις 4 Ιουνίου 1996, για να συντριφθεί 36 δευτερόλεπτα αργότερα λόγω ενός σφάλματος στο λογισμικό ελέγχου, και το Pentium FDIV bug το οποίο ήταν ένα σφάλμα που παρουσίασε ο επεξεργαστής της Intel κατά την διαίρεση δεκαδικών ψηφίων δίνοντας λανθασμένα αποτελέσματα, που ανάγκασε την εταιρεία να αποσύρει όλους τους ελαττωματικούς επεξεργαστές και να επωμιστεί τεράστιες ζημιές.

Τις δύο τελευταίες δεκαετίες, η περιοχή των τυπικών μεθόδων για το σχεδιασμό και την ανάλυση συστημάτων έχει αναπτυχθεί και ωριμάσει σημαντικά και έχει σημειώσει μεγάλη επιτυχία τόσο στην ανάπτυξη θεωρητικών μεθόδων περιγραφής και ανάλυσης συστημάτων όσο και στην παροχή μεθοδολογιών και πρακτικών εργαλείων για τους σκοπούς αυτούς. Ένα εργαλείο που παρέχει δυνατότητες ελέγχου και επαλήθευσης μοντέλων είναι το SPIN [3], το οποίο χρησιμοποιεί την γλώσσα επαλήθευσης μοντέλων Promela (Process Meta Language) [3]. Αυτό είναι και το εργαλείο που θα χρησιμοποιηθεί και θα αξιολογηθεί κατά την

εκπόνηση της συγκεκριμένης Ατομικής Διπλωματικής Εργασίας στο πεδίο των πολυπύρηνων συστημάτων.
Συγκεκριμένα, οι στόχοι της εργασίας είναι οι εξής:

- Η γνωριμία και η εξοικείωση με το εργαλείο SPIN.
- Η μελέτη του πεδίου των πολυπύρηνων συστημάτων.
- Η τυπική ανάλυση επιλεγμένου αλγορίθμου που έχει προταθεί στο πεδίο αυτό μέσω του εργαλείου SPIN.
- Η αξιολόγηση των δυνατοτήτων που παρέχει το εργαλείο SPIN για το πεδίο υπό μελέτη και ιδιαίτερα της χρονική λογικής LTL (Linear Temporal Logic) [1] για τον σκοπό αυτό.

1.2 Μεθοδολογία Υλοποίησης της Μελέτης

Για την εκπόνηση της εργασίας αυτής ακολουθήθηκε η εξής μεθοδολογία: Αρχικά μελετήθηκαν κάποιες εισαγωγικές έννοιες καθώς επίσης και τα εργαλεία με τα οποία ασχοληθήκαμε. Σε πρώτο στάδιο μελετήθηκε η γλώσσα Promela στην οποία θα γίνει η υλοποίηση. Χρησιμοποιήθηκαν δυο βιβλία [2, 3] μέσα από τα οποία κατανοήθηκαν οι βασικές έννοιες για την δημιουργία μοντέλων στην Promela, τις διάφορες δομές δεδομένων που υποστηρίζονται, τους τρόπους επικοινωνίας μεταξύ διεργασιών στον παράλληλο προγραμματισμό, καθώς επίσης και τους τρόπους επαλήθευσης μοντέλων που παρέχει το εργαλείο SPIN. Μέσα από τα βιβλία αυτά έγινε και μια πρώτη επαφή με το γραφικό περιβάλλον του XSPIN και με τις διάφορες λειτουργίες και επιλογές που παρέχει. Ακολούθως υλοποιήθηκαν σε κώδικα διάφορες ασκήσεις που είχαν να κάνουν με παράλληλη επεξεργασία και επαληθεύθηκε η ορθότητά τους με το εργαλείο SPIN. Με αυτό τον τρόπο έγινε μια καλύτερη εξοικείωση με τη γλώσσα και με το εργαλείο.

Σε δεύτερη φάση μελετήθηκε η χρονική λογική LTL μέσα από τις σημάνσεις του μαθήματος ΕΠΛ412 – Λογική στην Πληροφορική [1], και από κάποια κεφάλαια με παραδείγματα που υπήρχαν στα δυο βιβλία [2 και 3] . Για την καλύτερη κατανόηση και για συνδυασμό με τα προηγούμενα, υλοποιήθηκαν κάποιοι αλγόριθμοι αμοιβαίου αποκλεισμού όπως με σηματοφόρους (semaphores), με μεταβίβαση συμβόλου (token passing), με ανταλλαγή οδηγιών (exchange instructions), με οδηγίες κλειδώματος (lock instructions) και με οδηγίες

‘δοκίμασε και θέσε’ (test and set instructions). Στους αλγόριθμους αυτούς (οι οποίοι μελετήθηκαν από το βιβλίο συντρέχοντος προγραμματισμού [4]), έγινε έλεγχος για ικανοποίηση διάφορων ιδιοτήτων όπως π.χ. αδιεξόδων (deadlock), λιμού (Starvation), αδύναμης δικαιοσύνης (weak fairness) κ.λπ. Έπειτα μελετήθηκαν κάποια άρθρα [5, 6, 7, 8, 9] στα οποία γινόταν περιγραφή διάφορων τρόπων με τους οποίους ερευνητές έχουν χειριστεί τον έλεγχο μοντέλων σε πραγματικά συστήματα, και το επίπεδο αφαιρετικότητας που χρησιμοποιούν για την μοντελοποίηση συστημάτων και τη συγγραφή κώδικα Promela.

Ακολούθως, έγινε μια γενική μελέτη στο πεδίο των πολυπύρηνων επεξεργαστών μέσα από κάποια άρθρα και εγκυκλοπαίδειες [10,11,12,13], με έμφαση στο πρόβλημα συνοχής στις τοπικές μνήμες (cache) των πολυπύρηνων επεξεργαστών που είναι συνδεδεμένοι σε τοπολογία δακτυλίου. Αρχικά έγινε μια ανάλυση για τις ανάγκες αλλά και τα στάδια που οδήγησαν τελικά στην δημιουργία των πολυπύρηνων επεξεργαστών, μελετήθηκαν διάφορες αρχιτεκτονικές όπως επεξεργαστές με κοινή ή ξεχωριστή τοπική μνήμη (cache) και στην συνέχεια αναλύθηκαν τα πλεονεκτήματα αλλά και τα μειονεκτήματα τους. Έπειτα κατανοήθηκε η έννοια της συνοχής στις μνήμες των πολυπύρηνων επεξεργαστών, ποιες συνθήκες πρέπει να ικανοποιούνται για να υπάρχει συνοχή, και με ποιους τρόπους μπορούμε να επιβάλουμε συνοχή σε πολυπύρηνους επεξεργαστές. Ακολούθως δόθηκε έμφαση στους τρόπους που επικοινωνούν οι επεξεργαστές που βρίσκονται συνδεδεμένοι σε τοπολογία δακτυλίου και σε αλγόριθμους που επιβάλλουν συνοχή στις μνήμες τους. Μελετήθηκαν κάποιοι αλγόριθμοι [14, 15] και αποφασίσθηκε όπως υλοποιηθεί ο αλγόριθμος Greedy-Order που επιβάλει συνοχή στις τοπικές μνήμες επεξεργαστών που είναι ενωμένη σε δακτύλιο μονής κατεύθυνσης. Αφού υλοποιήθηκε ο αλγόριθμος στη γλώσσα Promela, έγινε η επαλήθευση του στο εργαλείο XSPIN και τέλος εξετάστηκαν κάποιες ιδιότητες χρονικής λογικής που αφορούσαν την ορθότητα του αλγορίθμου, και οι οποίες θα εξηγηθούν σε επόμενο κεφάλαιο.

Μέσω της εργασίας αυτής, είχα την ευκαιρία να μάθω το πόσο σημαντικός είναι ο μοντελο-έλεγχος συστημάτων και αλγορίθμων, τα πλεονεκτήματα αλλά και μειονεκτήματα που έχει και να γνωρίσω ένα δυνατό εργαλείο ελέγχου και επαλήθευσης μοντέλων, το SPIN, και της γλώσσας Promela που χρησιμοποιεί για την υλοποίηση μοντέλων. Μέσω των άρθρων που μελέτησα [5,6,7,8,9], κατανόησα τους τρόπους που μοντελοποιούν οι ειδικοί μεγάλα συστήματα και πρωτόκολλα σε αυτό το εργαλείο. Κατάφερα να αποκτήσω κάποια πείρα

στην μοντελοποίηση αλγορίθμων στην γλώσσα Promela, και μέσω των δυσκολιών που αντιμετώπισα και επίλυσα, έμαθα τις δυνατότητες που προσφέρει το εργαλείο SPIN. Επίσης το πεδίο των πολυπύρηνων επεξεργαστών που επιλέχθηκε για μελέτη, και από το οποίο επιλέχθηκε ο αλγόριθμος που υλοποιήθηκε, μου έδωσε την ευκαιρία να μελετήσω σε βάθος τρόπους με τους οποίους μπορεί να επιβληθεί συνοχή μνήμης σε πολυπύρηνους επεξεργαστές που είναι συνδεδεμένοι σε τοπολογία δακτυλίου.

1.3 Δομή της Διπλωματικής

Η διπλωματική αυτή χωρίζεται σε έξι κεφάλαια με την ακόλουθη δομή:

Στο Κεφάλαιο 2 εξηγείται η έννοια του μοντελο-ελέγχου και παρατίθενται τα πλεονεκτήματα και τα μειονεκτήματα του. Γίνεται μια εισαγωγή στο εργαλείο XSPIN και τις λειτουργίες του, και περιγράφεται η γλώσσα Promela την οποία χρησιμοποιεί για την δημιουργία μοντέλων. Τέλος γίνεται μια περιγραφή της χρονικής λογικής και των ιδιοτήτων που μπορούμε να ελέγξουμε με αυτήν.

Το Κεφάλαιο 3 αναφέρεται αποκλειστικά στους πολυπύρηνους επεξεργαστές, ξεκινώντας με μια εισαγωγή, και ακολούθως περιγράφεται η εξελικτική πορεία από τους μονοπύρηνους επεξεργαστές στους πολυπύρηνους. Στη συνέχεια αναφέρονται τα πλεονεκτήματα και τα μειονεκτήματα τους.

Στο Κεφάλαιο 4 εξηγείται η έννοια της συνοχής της μνήμης σε πολυπύρηνους επεξεργαστές, περιγράφεται η τοπολογία δακτυλίου, και πως μπορεί να επιβληθεί η συνοχή μνήμης σε πολυπύρηνους επεξεργαστές που είναι συνδεδεμένοι σε μια τέτοια τοπολογία.

Στο Κεφάλαιο 5 περιγράφεται ο αλγόριθμος Greedy Order [15] που αφορά την συνοχή μνήμης σε πολυπύρηνους επεξεργαστές που είναι συνδεδεμένοι σε τοπολογία δακτυλίου, περιγράφεται ο τρόπος λειτουργίας του αλλά και ο τρόπος υλοποίησης του και παρουσιάζονται τα αποτελέσματα επαλήθευσης του.

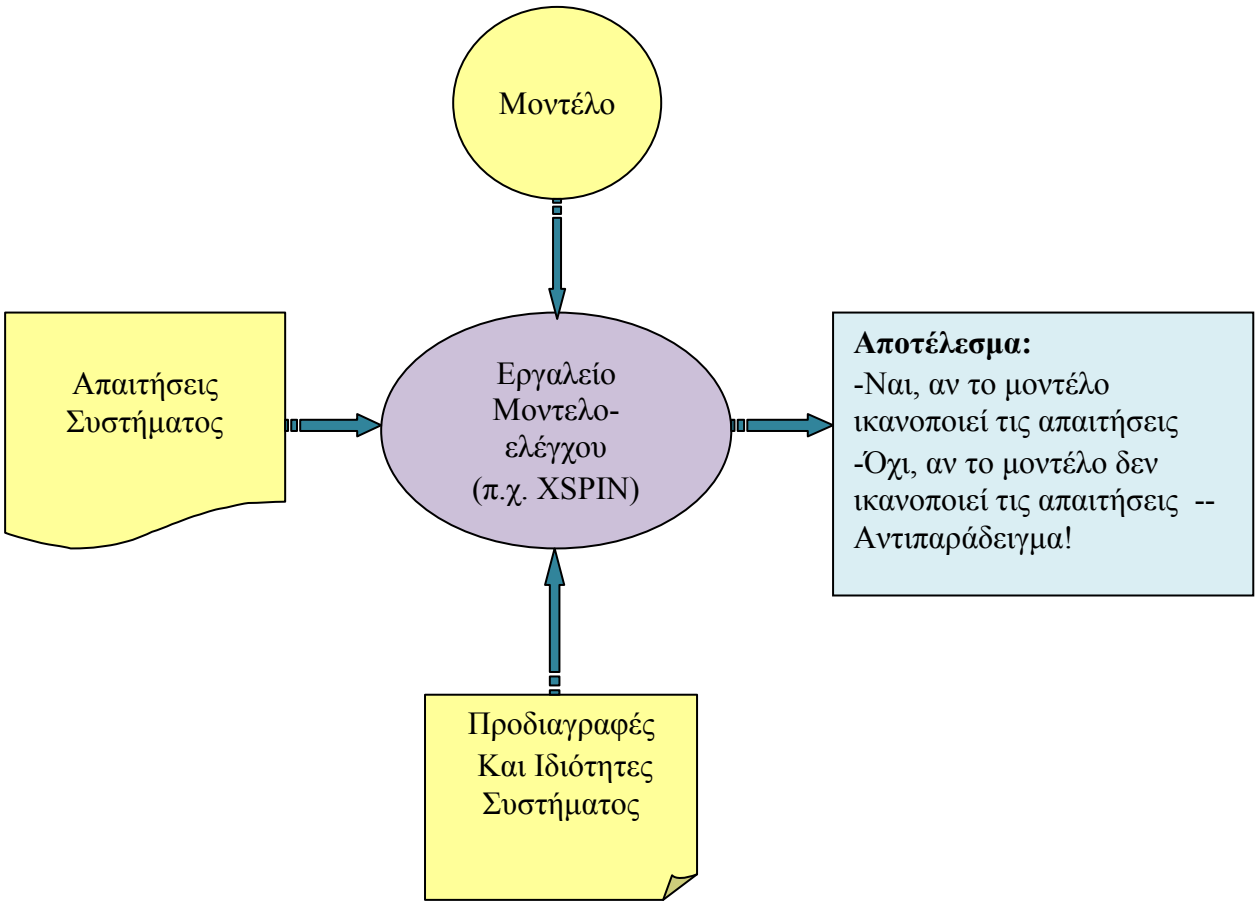
Το Κεφάλαιο 6 αναφέρεται σε συμπεράσματα και εντυπώσεις που αποκόμισα μέσω της επαφής μου με το εργαλείο SPIN.

Κεφάλαιο 2

Υπόβαθρο

2.1 Τι είναι μοντελο-έλεγχος

Τα τελευταία χρόνια, η επιστήμη έχει σημειώσει δραματική πρόοδο στο πεδίο της πληροφορικής, και ειδικότερα στην ανάπτυξη εργαλείων και τεχνικών για επαλήθευση των απαιτήσεων και του σχεδιασμού συστημάτων. Ο μοντελο-έλεγχος [1] είναι ίσως η πιο επιτυχημένη αυτοματοποιημένη τεχνική επαλήθευσης απαιτήσεων για συστήματα με πεπερασμένο αριθμό καταστάσεων. Καθορίζει εαν ένα σύνολο από ιδιότητες ικανοποιούνται σε κάποιο μοντέλο του συστήματος σε όλες τις δυνατές εκτελέσεις του, και σε αντίθετη περίπτωση δημιουργεί ένα αντιπαράδειγμα ευκολύνοντας τον προγραμματιστή να εντοπίσει το λάθος και να το διορθώσει. Η ιδέα που κρύβεται πίσω από τον μοντελο-έλεγχο είναι ότι εξασφαλίζουμε με την επαλήθευση του μοντέλου, ότι το σύστημα που θα αναπτυχθεί ικανοποιεί όλες τις απαιτήσεις που θα έπρεπε, και έτσι αυξάνεται η εμπιστοσύνη μας πως το ολοκληρωμένο σύστημα θα είναι ορθό. Όπως φαίνεται και από την Εικόνα 2.1, ένα εργαλείο μοντελο-ελέγχου δέχεται σαν είσοδο τις απαιτήσεις και το μοντέλο του συστήματος, και τις προδιαγραφές και ιδιότητες που πρέπει αυτό να ικανοποιεί. Σαν έξοδο από το εργαλείο έχουμε θετική απάντηση αν το σύστημα επαληθεύτηκε χωρίς λάθη, και αρνητική αν εντοπίστηκε κάποιο λάθος κατά την επαλήθευση. Σε αυτή την περίπτωση, δημιουργείται και κάποιο αντιπαράδειγμα που δείχνει που ακριβώς είναι το λάθος. Ένα εργαλείο που παρέχει αυτές τις δυνατότητες είναι το SPIN [3], με το οποίο θα ασχοληθούμε στην εργασία αυτή.



Εικόνα 2.1: *Προσέγγιση μοντελο-ελέγχου*

2.2 Πλεονεκτήματα – Μειονεκτήματα Μοντελο-ελέγχου

Ο μοντελο-έλεγχος είναι μια επαναστατική μέθοδος για την αυτοματοποιημένη επαλήθευση συστημάτων η οποία βασίζεται σε συστηματικό έλεγχο του συνόλου καταστάσεων ενός συστήματος. Κατά τον μοντελο-έλεγχο επιβεβαιώνεται ότι το μοντέλο και κατ' επέκταση το σύστημα ικανοποιεί τις ιδιότητες και απαιτήσεις που πρέπει, και οι προγραμματιστές μπορούν να προχωρήσουν στην υλοποίηση του. Σε αντίθετη περίπτωση, εντοπίζονται προβλήματα και λάθη που πολύ πιθανόν θα συναντηθούν κατά το στάδιο της υλοποίησης, ή ακόμα να γίνουν αντιληπτά μετά την ολοκλήρωση του. Ως αποτέλεσμα του εντοπισμού τέτοιων λαθών σε πρώιμο στάδιο, είναι η εξοικονόμηση τεράστιων κεφαλαίων από τις εταιρείες, αφού είναι γνωστό πως όσο πιο σύντομα βρεθεί το λάθος κατά την ανάπτυξη ενός συστήματος, τόσο πιο οικονομική είναι και η αποσφαλμάτωση του. Για παράδειγμα το σφάλμα FDIV που εντοπίστηκε στους επεξεργαστές της Intel μετά την κυκλοφορία τους στην αγορά, επέφερε ζημιές στην εταιρεία περίπου 475 εκατομμύρια δολάρια. Για αυτό το λόγο το ενδιαφέρον της βιομηχανίας για τον μοντελο-έλεγχο συνεχώς αυξάνεται. Άλλα πλεονεκτήματα του μοντελο-ελέγχου είναι το γεγονός ότι μπορεί να εφαρμοστεί σε πολλά συστήματα, για παράδειγμα σε λογισμικά συστήματα, σε πρωτόκολλα ή ακόμα και σε υλικό.

Ακόμα ένα πλεονέκτημα είναι ότι υπάρχουν έτοιμα και αξιόπιστα εργαλεία (π.χ. SPIN) τα οποία μπορούν να ελέγξουν όλες τις δυνατές καταστάσεις ενός μοντέλου και όταν εντοπίσουν λάθη να δημιουργήσουν κάποιο αντιπαράδειγμα, κάνοντας την ζωή του προγραμματιστή πιο εύκολη.

Ένα από τα μειονεκτήματα του μοντελο-ελέγχου είναι ότι ασχολείται κυρίως με την συμπεριφορά συστημάτων που έχουν πολύπλοκη ροή, και όχι με την επεξεργασία δεδομένων. Το κυριότερο του μειονέκτημα όμως εστιάζεται στην διαχείριση του χώρου.
Καθώς κατά τον μοντελο-έλεγχο διενεργείται για εξαντλητική ανάλυση όλων των καταστάσεων ενός συστήματος η αποθήκευση των καταστάσεων αυτών απαιτεί μεγάλη ποσότητα μνήμης. Η πολυπλοκότητα ενός μοντέλου μπορεί να οδηγήσει στο φαινόμενο της έκρηξης καταστάσεων (state space explosion), αφού η εισαγωγή κάθε καινούριας μεταβλητής δυνατόν να αυξήσει τον αριθμό των καταστάσεων εκθετικά. Και αυτό οδηγεί σε ένα άλλο μειονέκτημα. Όταν σχεδιάζουμε μοντέλα, πρέπει να λαμβάνουμε υπόψη αυτό το πρόβλημα αλλά και το γεγονός ότι τα αποτελέσματα που θα πάρουμε θα είναι ανάλογα με την ποιότητα του μοντέλου. Αλλά ο σχεδιασμός κατάλληλων μοντέλων απαιτεί πείρα έτσι ώστε ένα σύστημα να περιγράφεται όσο το δυνατό πιο απλά, καλύπτοντας όμως όλες τις λειτουργίες και ιδιότητες του.

2.3 Εργαλείο XSPIN

Το εργαλείο SPIN (= Simple Promela Interpreter) [3] είναι ένα δυνατό εργαλείο μοντελο-ελέγχου, το οποίο χρησιμοποιεί την γλώσσα επαλήθευσης μοντέλων Promela για την αναπαράσταση του μοντέλου κάποιου συστήματος. Το SPIN δουλεύει εκτελώντας εντολές από κάποιο τερματικό, και αυτό αναγκάζει τον χρήστη να θυμάται τις διάφορες εντολές προτού τρέξει κάποια προσομοίωση ή επαλήθευση. Για τον λόγο αυτό δημιουργήθηκε το XSPIN [3] το οποίο παρέχει γραφικό περιβάλλον μέσω του οποίου μπορούν να εκτελεστούν όλες οι λειτουργίες του SPIN, κάνοντας τη δουλειά του προγραμματιστή πιο εύκολη. Το εργαλείο αυτό παρέχει δυνατότητες προσομοίωσης και επαλήθευσης μοντέλων, επαλήθευσης ιδιοτήτων χρονικής λογικής, παρουσίαση των αυτομάτων για κάθε διεργασία του μοντέλου, παρακολούθηση της αλλαγής αλλά και των τελικών τιμών των μεταβλητών του μοντέλου σε κάποια προσομοίωση και γραφική αναπαράσταση των καναλιών που χρησιμοποιούνται για την επικοινωνία των διεργασιών του μοντέλου.

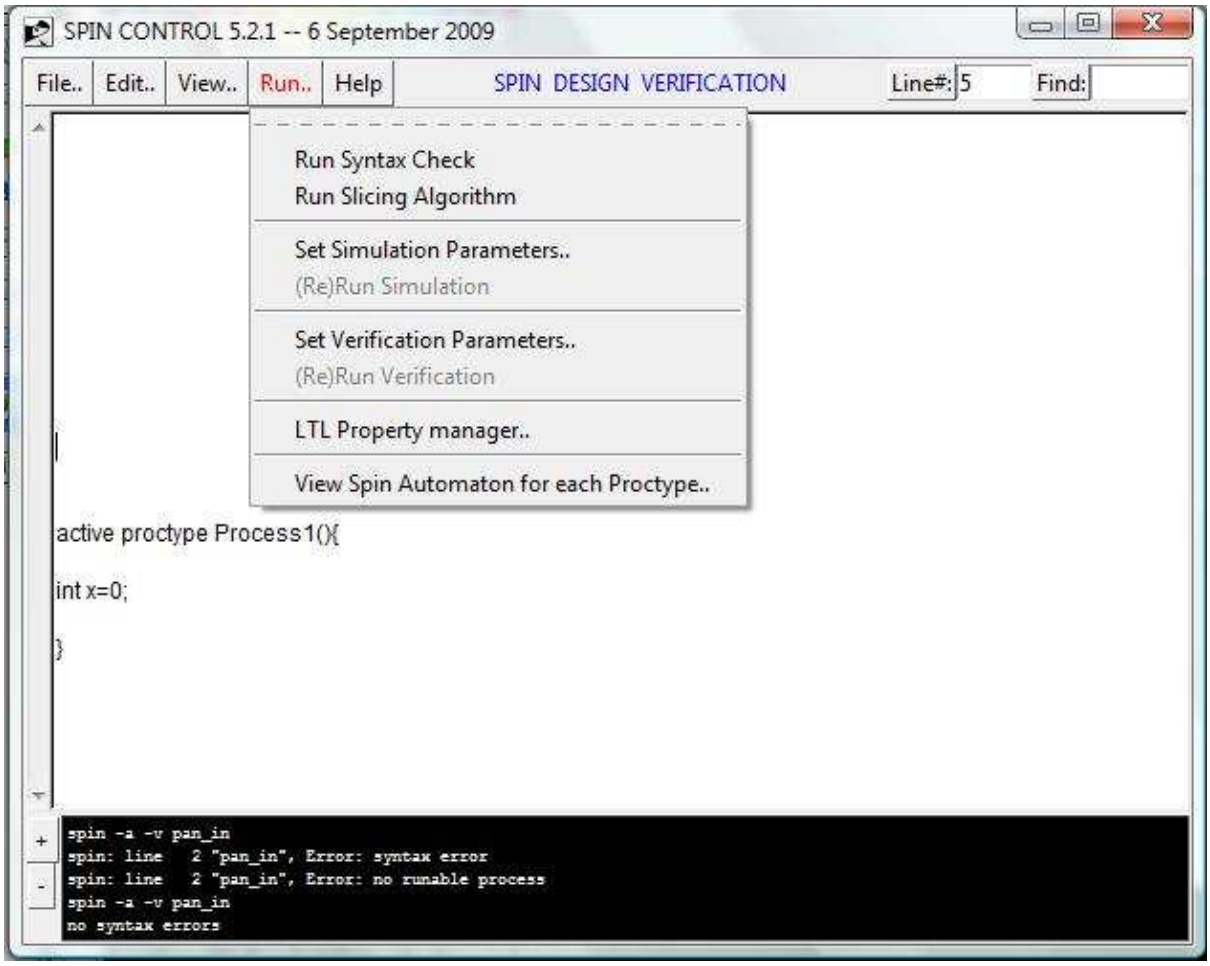
Για να δουλέψει κάποιος με το XSPIN, αρχικά πρέπει να γράψει τον κώδικα ο οποίος μοντελοποιεί ένα παράλληλο αλγόριθμο ή σύστημα. Ακολούθως ελέγχει τον κώδικα του για συντακτικά λάθη και έπειτα μπορεί να τρέξει μια προσομοίωση για να επιβεβαιώσει ότι ο κώδικας του συμπεριφέρεται όπως πρέπει. Αφού γίνει αυτό και το μοντέλο δείχνει να συμπεριφέρεται σωστά, μπορεί να προχωρήσει στην επόμενη φάση κατά την οποία γίνεται επαλήθευση του μοντέλου, η οποία μπορεί να γίνει με συνδυασμούς επιλογών αναλόγως του τι θέλουμε να ελέγξουμε. Σε περίπτωση που κατά την επαλήθευση προκύψει λάθος, τότε ακολουθούμε την καθοδηγημένη προσομοίωση που δημιούργησε το SPIN, έτσι ώστε να εντοπίσουμε το λάθος, να το διορθώσουμε και να επαναλάβουμε την επαλήθευση.

Το SPIN όμως έχει και κάποιες αυστηρές απαιτήσεις ειδικά στον παράγοντα χώρου. Στον έλεγχο μοντέλων οι απαιτήσεις σε χώρο (αριθμός καταστάσεων), είναι πολύ μεγαλύτερες από αυτές του κοινού προγραμματισμού. Ειδικότερα πρέπει να ληφθούν υπόψη κάποιοι παράγοντες όπως ο αριθμός καταστάσεων, όπου υπάρχει κίνδυνος εμφάνισης του φαινομένου της έκρηξης καταστάσεων κατά το οποίο ο αριθμός των καταστάσεων που παράγει μια εφαρμογή αυξάνεται συνεχώς φτάνοντας σε τεράστια νούμερα. Για τον λόγο αυτό πρέπει να γίνεται προσπάθεια να διατηρείται όσο το δυνατό μικρότερος ο αριθμός καταστάσεων. Άλλος παράγοντας είναι το μέγεθος της στοίβας αναζήτησης, όπου πρέπει να διατηρείται μικρή, και αυτό μπορεί να γίνει χρησιμοποιώντας την ιδιότητα επαλήθευσης State Compression η οποία μειώνει τον αριθμό των καταστάσεων προς επαλήθευση, χρησιμοποιώντας μόνο τις πιο ουσιαστικές του μοντέλου. Τέλος το μέγεθος του χρόνου επαλήθευσης μας δείχνει αν το μοντέλο χρειάζεται κάποιες βελτιστοποιήσεις (σε περίπτωση που ο χρόνος είναι μεγάλος τότε το μοντέλο πρέπει να βελτιστοποιηθεί).

2.3.1 Προσομοίωση στο XSPIN

Το εργαλείο XSPIN αποτελεί το γραφικό περιβάλλον του SPIN (Εικόνα 2.2). Αφού έχουμε γράψει τον κώδικα κάποιου μοντέλου, το πρώτο βήμα που θα ακολουθήσουμε είναι να ελέγξουμε για συντακτικά λάθη. Αυτό είναι εφικτό μέσω του μενού εντολών επιλέγοντας την εντολή "Run Syntax check" που βρίσκεται κάτω από την επιλογή "Run". Στο μαύρο πλαίσιο φαίνεται το αποτέλεσμα του ελέγχου (no syntax errors), καθώς επίσης και η εντολή του SPIN που εκτελέστηκε (spin -a –v pan_in). Σε περίπτωση που υπάρχουν συντακτικά λάθη, τότε ο

προγραμματιστής θα ειδοποιηθεί από ένα αναδυόμενο παράθυρο που θα αναφέρει το είδος του λάθους καθώς και την γραμμή. Αυτό θα είναι ορατό και στο μαύρο πλαίσιο μηνυμάτων.



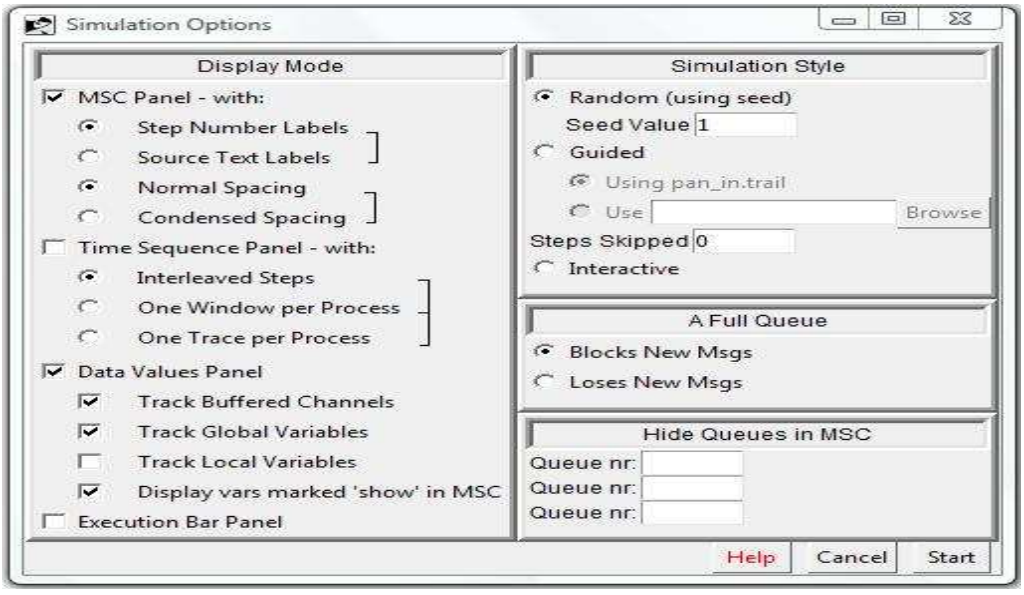
Εικόνα 2.2 *Γραφικό περιβάλλον XSPIN*

Το επόμενο βήμα είναι να προχωρήσουμε στην προσομοίωση του μοντέλου, μέσω της επιλογής "Set Simulation Parameters" που βρίσκεται κάτω από την επιλογή "Run" (Εικόνα 2.2). Αφού το κάνουμε αυτό, θα εμφανιστεί ένα νέο παράθυρο (Εικόνα 2.3), στο οποίο δίνεται η δυνατότητα στον προγραμματιστή να επιλέξει από ένα σύνολο παραμέτρων, ανάλογα με το τι θέλει να δει κατά την προσομοίωση. Όπως τονίζει και το βιβλίο [2], οι αρχάριοι προγραμματιστές θα πρέπει να αποφεύγουν τις αλλαγές στις παραμέτρους αυτές, και θα πρέπει να προτιμούν τις προσομοιώσεις μοντέλων με τις προεπιλεγμένες επιλογές.

Στο αριστερό μισό της Εικόνας 2.3 (Display Mode), παρέχεται η δυνατότητα στο χρήστη να διαμορφώσει το πώς θα παρουσιάζεται το αποτέλεσμα της προσομοίωσης, ανάλογα με το τι θέλει να εξετάσει. Η πρώτη επιλογή (MSC – Message Sequence Chart Panel), παρέχει στον

προγραμματιστή να διαμορφώσει την γραφική αναπαράσταση των μηνυμάτων που ανταλλάσσουν μεταξύ τους οι διεργασίες κατά την προσομοίωση. Στην δεύτερη επιλογή (Time Sequence Panel), ο προγραμματιστής μπορεί να παρακολουθήσει βήμα προς βήμα την προσομοίωση και την εκτέλεση των εντολών, και να επιλέξει αν θέλει κάθε διεργασία να παρουσιάζεται σε ξεχωριστό παράθυρο. Η τρίτη επιλογή αφήνει τον προγραμματιστή να επιλέξει ποιες μεταβλητές θέλει να παρακολουθήσει.

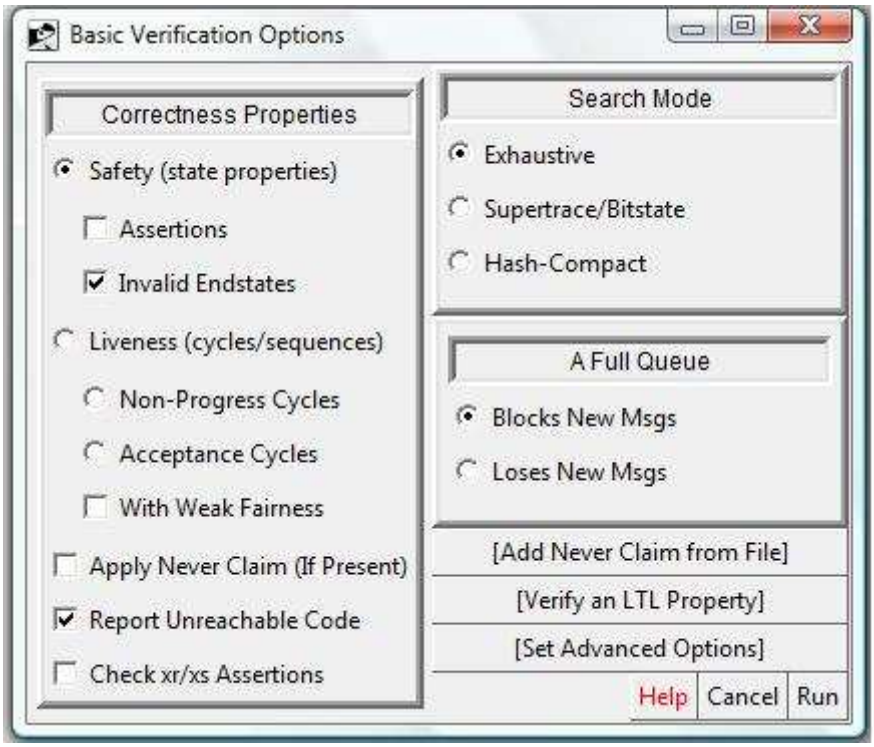
Στο δεξιό πάνω μέρος της εικόνας (Simulation Style) παρουσιάζονται τρείς διαφορετικοί τρόποι προσομοίωσης. Στην πρώτη επιλογή (Random), οι αποφάσεις κατά την διάρκεια της προσομοίωσης παίρνονται με εντελώς τυχαία σειρά. Στην δεύτερη επιλογή, (Guided), ο προγραμματιστής μπορεί να βρει και να ακολουθήσει το μονοπάτι στο οποίο εντοπίστηκε κάποιο λάθος και να καταλάβει σε ποιο σημείο ακριβώς εμφανίζεται το συγκεκριμένο λάθος. Στην τελευταία επιλογή (Interactive), δίνεται η δυνατότητα στον προγραμματιστή να καθοδηγήσει την εκτέλεση της προσομοίωσης, επιλέγοντας κάθε φορά την επόμενη εκτελέσιμη εντολή. Στην τελευταία επιλογή (Full Queue), σε περίπτωση που σταλεί κάποιο μήνυμα σε ένα γεμάτο buffered κανάλι (θα εξηγηθεί στην επόμενη ενότητα), τότε δίνεται η δυνατότητα στον προγραμματιστή να επιλέξει αν αυτό το μήνυμα θα χαθεί (Loses New Msgs), ή αν θα μπλοκάρει (blocks New Msgs) η διεργασία που το στέλνει γιατί δεν μπορεί ο παραλήπτης να το λάβει.



Εικόνα 2.3 *Επιλογές Προσομοίωσης*

2.3.2 Επαλήθευση στο XSPIN

Μετά την προσομοίωση, το επόμενο βήμα είναι η επαλήθευση του μοντέλου. Αυτό μπορεί να επιτευχθεί από το μενού Run επιλέγοντας την εντολή Set Verification Parameters. Η επιλογή αυτή θα εμφανίσει ένα νέο παράθυρο (Εικόνα 2.4), δίνοντας στον προγραμματιστή την ευχέρεια να καθορίσει τις παραμέτρους επαλήθευσης που θα χρησιμοποιήσει κατά την επαλήθευση του μοντέλου του.



Εικόνα 2.4 *Παράμετροι επαλήθευσης*

Στο αριστερό μισό του παραθύρου φαίνονται οι παράμετροι ορθότητας (Correctness Properties). Η πρώτη επιλογή αφορά ιδιότητες ασφαλείας, δηλαδή μπορούμε να επιβάλλουμε να γίνει έλεγχος για λανθασμένες τελικές καταστάσεις (π.χ. Deadlocks), ή για παραβιάσεις συνθηκών που βρίσκονται σε μια εντολή assert(«Boolean condition»). Για να επιτραπεί ο έλεγχος της συνθήκης assert(), πρέπει φυσικά να υπάρχει τουλάχιστο μια τέτοια εντολή στον κώδικα του μοντέλου, αλλιώς το εργαλείο θα βγάλει μήνυμα προειδοποίησης ότι δεν μπορεί να γίνει τέτοιος έλεγχος αφού δεν εντοπίστηκε καμία εντολή assert().

Ακολουθούν οι ιδιότητες βιωσιμότητας (Liveness) οι οποίες χωρίζονται σε Non-Progress Cycles, Acceptance Cycles και Weak fairness. Στην ιδιότητα Non-Progress Cycles, ο

προγραμματιστής στιγματίζει κάποια σημεία του κώδικα του μοντέλου που θέλει να ελέγξει για πρόοδο, με λέξεις που ξεκινούν με το πρόθεμα “progress”, και το εργαλείο θα ελέγξει αν όντως η εκτέλεση περνά από το στιγματισμένο σημείο έτσι ώστε να υπάρχει πρόοδος στο μοντέλο. Στην δεύτερη ιδιότητα (Acceptance Cycles), ο προγραμματιστής πρέπει και πάλι να προσθέσει ετικέτες που ξεκινούν με το πρόθεμα “accept” στα σημεία του κώδικα που θέλει να ελέγξει. Με αυτό τον τρόπο ζητείται από το εργαλείο να βρει αν υπάρχει εκτέλεση που να περνά από μια ετικέτα accept άπειρες φορές. Η ιδιότητα weak fairness ελέγχει την ιδιότητα ότι σε όλες τις εκτελέσεις του μοντέλου, όταν κάποια διεργασία κάνει αίτηση για να εξυπηρετηθεί, τότε όντως κάποτε θα εξυπηρετηθεί.

Η επόμενη ιδιότητα (Apply never claim – if present) απαιτεί από τον προγραμματιστή να έχει γράψει μια πρόταση never claim, η οποία δεν θέλει να επαληθευτεί ποτέ. Αν δεν υπάρχει κάποια πρόταση never claim και ο προγραμματιστής επιχειρήσει να επαληθεύσει το μοντέλο με αυτή την επιλογή, τότε το εργαλείο θα εμφανίσει προειδοποιητικό μήνυμα για το γεγονός ότι δεν βρέθηκε η αντίστοιχη πρόταση. Το check xr/xs assertions, αφορά assertions για κανάλια. Το xr δηλώνει ότι η διεργασία που το εκτελεί έχει αποκλειστικά δικαιώματα παραλαβής μηνυμάτων από το κανάλι στο οποίο αναφέρεται (π.χ. xr channel1),και το xs αντίστοιχα δηλώνει ότι η διεργασία που το εκτελεί έχει αποκλειστικά δικαιώματα αποστολής στο κανάλι στο οποίο αναφέρεται (π.χ. xs channel1) .

Στο δεξιό κομμάτι του παραθύρου, υπάρχουν τρία είδη αναζήτησης. Το πρώτο (Exhaustive) αναφέρεται σε εξαντλητική αναζήτηση στο χώρο καταστάσεων του μοντέλου. Το δεύτερο είδος (Supertrace/Bitstate) χρησιμοποιείται σε περιπτώσεις που ο χώρος καταστάσεων δεν προσφέρεται για εξαντλητική αναζήτηση. Δηλαδή δεν γίνεται αναζήτηση σε ολόκληρο το σύνολο καταστάσεων του μοντέλου. Το τρίτο είδος (Hash-Compact) συμπιέζει το χώρο καταστάσεων, εξοικονομώντας μνήμη, σε βάρος όμως του χρόνου επαλήθευσης (λόγω της συμπίεσης). Ο προγραμματιστής μπορεί να ρυθμίσει και πιο προχωρημένες παραμέτρους για την επαλήθευση του μοντέλου, όπως για παράδειγμα το μέγεθος της φυσικής μνήμης του υπολογιστή, μέσω της επιλογής Set Advanced Options. Μπορεί επίσης να ελέγξει μια ιδιότητα χρονική λογικής μέσω της επιλογής Verify an LTL Property.

2.4 Γλώσσα Επαλήθευσης Μοντέλων Promela

Η γλώσσα που χρησιμοποιείται για την διατύπωση μοντέλων στο SPIN είναι η γλώσσα Promela. Η γλώσσα αυτή έχει πολλά κοινά ως προς την σύνταξη της, με την γλώσσα προγραμματισμού C π.χ. έχει τους ίδιους τελεστές και πολλούς κοινούς τύπους δεδομένων. Τα προγράμματα σε αυτήν την γλώσσα αποτελούνται από ένα σύνολο διεργασιών, οι οποίες τρέχουν παράλληλα και μη ντετερμινιστικά. Κάθε διεργασία μπορεί να επικοινωνεί με τις υπόλοιπες διεργασίες του προγράμματος μέσω ανταλλαγής μηνυμάτων με την βοήθεια των καναλιών. Τα κανάλια είναι μια επιπλέον δομή δεδομένων που υποστηρίζει η γλώσσα αυτή και επιτρέπει στον προγραμματιστή να μεταφέρει δεδομένα από την μια διεργασία στη άλλη. Όπως και όλες οι γλώσσες προγραμματισμού, έτσι και η Promela, υποστηρίζει δομές επανάληψης (do::), δομές ελέγχου (if::) και συνάρτηση για τύπωμα στην οθόνη(printf();). Η αξιοσημείωτη διαφορά της γλώσσας αυτής με τις γλώσσες προγραμματισμού, είναι ότι δεν έχει συνάρτηση για είσοδο δεδομένων από τον χρήστη, για τον απλούστατο λόγο ότι δεν χρειάζεται! Σκοπός αυτής της γλώσσας είναι η περιγραφή μοντέλων με σκοπό την επαλήθευση τους και όχι ο υπολογισμός τιμών ή η αποθήκευση δεδομένων όπως γίνεται στις γλώσσες προγραμματισμού. Με τον όρο "επαλήθευση συστήματος", εννοούμε ότι οι ιδιότητες και οι συνθήκες που χαρακτηρίζουν το σύστημα, πρέπει να ισχύουν σε όλες τις δυνατές εκτελέσεις του συστήματος και να μην παραβιάζονται ποτέ. Για παράδειγμα, στο πρόβλημα του αμοιβαίου αποκλεισμού, πρέπει να ισχύει πάντα η ιδιότητα ότι αν κάποια διεργασία εκτελεί το κρίσιμο κομμάτι της, τότε καμιά άλλη διεργασία δεν μπορεί να εκτελεί το δικό της κρίσιμο κομμάτι την ίδια στιγμή. Η Promela υποστηρίζει την εντολή assert(), στην οποία δίνεται παράμετρος μια αμετάβλητη συνθήκη που πρέπει να ισχύει σε όλες τις εκτελέσεις του μοντέλου. Όταν τρέξουμε το πρόγραμμα και υπάρχει εκτέλεση που παραβιάζει αυτή την συνθήκη, τότε η εκτέλεση σταματά και δίνεται αντιπαράδειγμα μέσω του εργαλείου XSPIN στο οποίο περιγράφεται η περίπτωση στην οποία παραβιάζεται η συνθήκη.

Όπως έχει προαναφερθεί ένα πρόγραμμα γραμμένο στη Promela, αναπαρίσταται μέσω των διεργασιών(ή διαδικασιών) οι οποίες ορίζονται μέσα σε ένα πρόγραμμα, και οι οποίες μπορούν να επικοινωνήσουν μεταξύ τους μέσω των καναλιών. Κάθε δήλωση έχει δύο δυνατότητες, είτε να εκτελεστεί είτε να μπλοκαριστεί, αναγκάζοντας την συγκεκριμένη διεργασία να περιμένει μέχρι οι συνθήκες να επιτρέψουν την εκτέλεση της. Για παράδειγμα μια δήλωση μπορεί να μπλοκαριστεί από μια εντολή if εφόσον η συνθήκη του if είναι

ψευδής. Η δήλωση όμως αυτή μπορεί να είναι εκτελέσιμη από την στιγμή που κάποια άλλη διεργασία θα κάνει την συνθήκη του συγκεκριμένου if αληθή, επιτρέποντας στο σύστημα να εκτελέσει την πρώην μπλοκαρισμένη εντολή.

Κάθε διεργασία στην Promela ορίζεται με την λέξη proctype. Οι διεργασίες στην Promela εκτελούνται παράλληλα, και αν υπάρχουν περισσότερες από δυο εκτελέσιμες εντολές, τότε επιλέγεται μια τυχαία. Ακολουθεί ένα μικρό παράδειγμα δήλωσης μιας διεργασίας:

```
proctype Process1 (int Y){
Int X = 5;
X = Y;
}
```

Το όνομα της διεργασίας είναι Process1 η οποία παίρνει ως παράμετρο την μεταβλητή Y. Οι αγκύλες δηλώνουν τον χώρο δηλώσεων και ελέγχων της διεργασίας αυτής. Δηλώνεται επίσης μια τοπική μεταβλητή X τύπου integer η οποία κατά την δήλωση της παίρνει τιμή 5 ενώ στην συνέχεια της ανατίθεται η τιμή της μεταβλητής της παραμέτρου Y. Στο παράδειγμα που ακολουθεί δηλώνονται δυο διεργασίες και φαίνεται ο τρόπος που χρησιμοποιείται η δομή ελέγχου if:

```
Int  X = 2;
proctype Process1(){
if
::X==1->X=X+1;
fi;
}
proctype Process2(){
X=X-1;
}
```

Το βέλος αναπαριστά την σχέση ‘αιτία -> συνέπεια’. Στο πιο πάνω πρόγραμμα δηλώνεται μια καθολική (global) μεταβλητή η οποία αρχικοποιείται με 2. Η διεργασία Process1 μπορεί να εκτελέσει το πολύ δυο βήματα. Είτε να μπλοκαριστεί, είτε να αυξήσει την τιμή της X κατά ένα. Αν εκτελεστεί πρώτα η Process1 τότε θα μπλοκαριστεί και θα περιμένει την

Process2 να μειώσει την μεταβλητή X έτσι ώστε η συνθήκη X==1 να γίνει αληθής και να ξεμπλοκαριστεί.

Για να μπορέσει μια διεργασία να τρέξει, πρέπει με κάποιο τρόπο πρώτα να ενεργοποιηθεί. Υπάρχουν δυο τρόποι για να ενεργοποιηθεί μια διεργασία. Είτε με την εντολή run μέσω της διεργασίας init η οποία δεν χρειάζεται να ενεργοποιηθεί, είτε ενεργοποιείται από την αρχή με την εντολή active. Ακολουθεί παράδειγμα ενεργοποίησης διεργασιών και με τους δυο τρόπους:

```
init {
run Process1();
}

active proctype Process1(){
…
}
```

Όπως έχουμε αναφέρει, για να επικοινωνήσουν δυο διεργασίες χρειάζεται η αποστολή και παραλαβή μηνυμάτων. Αυτό μπορεί να γίνει μέσω των καναλιών. Ακολουθεί παράδειγμα δήλωσης καναλιού :

```
chan channel1 = [N] of {int}
channel1!4;
channel1?number;
```

Στο πιο πάνω παράδειγμα στην πρώτη γραμμή δηλώνουμε ένα κανάλι (buffered κανάλι) το οποίο μπορεί να περιέχει μέχρι και N διαφορετικά μηνύματα τα οποία αποτελούνται από μια μεταβλητή integer. Στην δεύτερη και Τρίτη γραμμή παρουσιάζεται ο τρόπος αποστολής και παραλαβής μηνυμάτων αντίστοιχα. Στο πιο κάτω παράδειγμα βλέπουμε πως ακριβώς μπορούν να επικοινωνήσουν δυο διεργασίες με κανάλια:

```
chan channel1 = [N] of {int}
proctype Process1(){
channel1!4;
```

```
}
proctype Process2(){
int number;
channel1?number;
}
```

Σε περίπτωση που εκτελεστεί πρώτα η διεργασία Process1, τότε θα στείλει το μήνυμα και ακολούθως θα το λάβει η Process2 αποθηκεύοντας τον αριθμό 4 στην τοπική της μεταβλητή number. Αν εκτελεστεί πρώτα η Process2, τότε θα περιμένει να λάβει μήνυμα από το κανάλι channel1, το οποίο είναι άδειο, και άρα η διεργασία θα μπλοκαριστεί, μέχρι να εκτελεστεί η Process1 και να στείλει το μήνυμα στο channel1, ξεμπλοκάροντας την Process2. Το κανάλι που χρησιμοποιήθηκε ονομάζεται ασύγχρονο κανάλι. Η Promela παρέχει και μια ειδική περίπτωση καναλιών, τα συγχρονισμένα κανάλια (rendez-vous channels) τα οποία δηλώνονται με ακριβώς τον ίδιο τρόπο αλλά έχουν χωρητικότητα 0 ([0]). Αυτό το είδος καναλιού, δεν μπορεί να αποθηκεύσει μηνύματα, και χρησιμοποιείται όταν θέλουμε τον αποστολέα να συγχρονιστεί με κάποια άλλη διεργασία, για να προχωρήσει σε κάποια άλλη ενέργεια.

Επίσης η Promela υποστηρίζει την επαναληπτική δομή do η οποία έχει την πιο κάτω σύνταξη:

```
do
:: (Y=1)->…
:: else ->break;
od;
```

Το else εκτελείτε μόνο σε περίπτωση που καμιά από τις άλλες συνθήκες δεν είναι αληθής. Μια άλλη σημαντική εντολή που υποστηρίζεται είναι η εντολή timeout. Αυτή η εντολή επιτρέπει στο σύστημα να ξεφύγει από τυχόν αδιέξοδο (deadlock). Σε περίπτωση που καμία εντολή του προγράμματος δεν μπορεί να εκτελεστεί, τότε ενεργοποιείται και εκτελείται η εντολή timeout η οποία παίρνει τιμή 1 όταν καμία άλλη εντολή στο σύστημα δεν μπορεί να εκτελεστεί και 0 σε όλες τις άλλες περιπτώσεις.

2.5 Ιδιότητες που μπορούμε να ελέγξουμε με το εργαλείο SPIN

Με το εργαλείο SPIN μπορούμε να ελέγξουμε κάποιες ιδιότητες που αφορούν ορισμούς εννοιών που μπορεί να συναντηθούν κατά τον έλεγχο ενός μοντέλου. Με άλλα λόγια ένα σύνολο προβλημάτων που μπορεί κάποιος να συναντήσει κατά τον έλεγχο ενός μοντέλου, τα οποία καλείται να επιλύσει.

Η πρώτη ιδιότητα είναι το αδιέξοδο (deadlock). Το αδιέξοδο είναι μια κατάσταση στην οποία μπορούν να φτάσουν δυο ή περισσότερες διεργασίες οι οποίες έχουν κάποια κοινή πηγή, όταν και ταυτόχρονα και οι δυο θελήσουν να έχουν πρόσβαση στη συγκεκριμένη πηγή, αποκλείοντας έτσι η μια την άλλη από το να πάρει την πηγή. Έτσι καταλήγουμε σε μια κατάσταση μπλοκαρίσματος και των δυο διεργασιών.

Η δεύτερη ιδιότητα αφορά την αδυναμία τερματισμού (livelock). Όταν μια διεργασία φτάσει σε κάποιο σημείο στο οποίο της είναι αδύνατο να τερματίσει με οποιοδήποτε τρόπο την εκτέλεση της, τότε αυτή η διεργασία έφτασε σε κατάσταση αδυναμίας τερματισμού. Σε μια τέτοια κατάσταση φτάνει μια διεργασία η οποία ενώ το κρίσιμο σημείο της ήδη ενασχολείται με μια εργασία, συνεχώς καταφθάνουν στην ουρά της επιπλέον απαιτήσεις για εργασίες, αφήνοντας της έτσι μηδενικό χρόνο για να καθαρίσει την ουρά της. Να σημειώσουμε ότι η διεργασία δεν μπλοκάρεται όπως στο αδιέξοδο, αλλά έχει μια άπειρη ποσότητα εργασίας να τελειώσει, μη μπορώντας να τερματίσει. Για παράδειγμα, αυτό μπορεί να συμβεί σε κάποιο πυρήνα όπου καταφθάνουν ακατάπαυστα πολλές απαιτήσεις. Έτσι όλη η υπολογιστική δύναμη του επεξεργαστή ξοδεύεται στην επεξεργασία των συνεχώς εισερχομένων αιτήσεων, αφήνοντας μηδενική υπολογιστική δύναμη για εκτέλεση της εργασίας που ήδη υπάρχει στον πυρήνα.

Μια άλλη ιδιότητα είναι οι ισχυρισμοί, οι οποίοι χρησιμοποιούνται με τις δηλώσεις never claim. Τέτοιοι ισχυρισμοί χρησιμοποιούνται για τον έλεγχο ύπαρξης μη επιθυμητών ή ακόμα και λανθασμένων καταστάσεων. Το SPIN επιστρέφει λάθος, δηλαδή βρήκε λανθασμένη κατάσταση, σε περίπτωση που η διεργασία τερματίζει σε κάποιο σημείο διαφορετικό από το τέλος της.Το never claim τερματίζει στο τέλος της διεργασίας.

Οι καταστάσεις αποδοχής (accept states), είναι μια άλλη ιδιότητα, κατά την οποία ο χρήστης μπορεί να δηλώσει ένα κομμάτι κώδικα ως κατάσταση αποδοχής, και αν η κατάσταση αυτή

εκτελεστεί άπειρες φορές, τότε το SPIN επιστρέφει λάθος. Αυτή η ιδιότητα παρέχει στον προγραμματιστή την δυνατότητα να κλείσει κάποιο κομμάτι κώδικα σε μια ετικέτα accept και έπειτα χρησιμοποιώντας το SPIN να ελέγξει για ένα άπειρο αριθμό κατά πόσο μια διεργασία εισέρχεται στην περιοχή του κώδικα που είναι κλεισμένος στην ετικέτα accept, και αν αυτό γίνει τότε το SPIN θα βγάλει λάθος.

Όπως έχει αναφερθεί και ποιο πάνω, η εντολή assert() χρησιμοποιείται σε περιπτώσεις που θέλουμε να ελέγχουμε ότι πάντα ισχύει κάποια συνθήκη. Στις παρενθέσεις περικλείεται μια έκφραση τύπου Boolean. Όταν ικανοποιείται η συνθήκη του assertion, τότε η ροή του προγράμματος δεν επηρεάζεται. Αν δεν ικανοποιείται έστω και σε μια εκτέλεση του προγράμματος, τότε θα έχουμε μήνυμα λάθους.

Επίσης υπάρχουν οι ιδιότητες ασφαλείας (Safety Properties) και βιωσιμότητας (Liveness Properties). Με την ιδιότητα ασφαλείας βεβαιωνόμαστε ότι κατά την εκτέλεση μιας διεργασίας δεν πέφτει ποτέ σε λάθος όσο ικανοποιείται η ορισμένη ιδιότητα. Με τις ιδιότητες βιωσιμότητας μπορούμε να αποδείξουμε ότι στο τέλος της διεργασίας θα ικανοποιούνται κάποιες συνθήκες που ορίζουμε εμείς .

2.6 Παράδειγμα αμοιβαίου αποκλεισμού σε Promela

Ακολουθεί ένα μικρό παράδειγμα (Παράδειγμα 2.1) υλοποίησης αλγορίθμου αμοιβαίου αποκλεισμού με σηματοφόρους (semaphores) [4] στην γλώσσα Promela, όπου φαίνεται η δομή των διεργασιών, των επαναληπτικών δομών, των δομών ελέγχου και πως γίνεται η δήλωση και χρησιμοποίηση των καναλιών. Το παράδειγμα χρησιμοποιεί μια σηματοφόρο που αντιπροσωπεύεται μέσω της μεταβλητής mutex. Όταν η μεταβλητή αυτή έχει τιμή 0(false), τότε σημαίνει πως κάποια διεργασία εκτελεί το κρίσιμο κομμάτι της και μπλοκάρει άλλες διεργασίες από του να μπουν και να εκτελέσουν και αυτές το δικό τους κρίσιμο κομμάτι. Όταν πάρει τιμή 1(true), τότε κάποια άλλη διεργασία δικαιούται να μπει να εκτελέσει το δικό της κρίσιμο κομμάτι κάνοντας την μεταβλητή 0 και αποκλείοντας και πάλι όλες τις άλλες μέχρι να τελειώσει. Σε αυτό το παράδειγμα οι διεργασίες P και V συγχρονίζουν το κρίσιμο κομμάτι των διεργασιών Process1 και Process2 έτσι ώστε να μην εκτελεστούν ποτέ και τα δυο μαζί την ίδια χρονική στιγμή. Η διεργασία P αντιστοιχεί στην κατάσταση όπου μια διεργασία περιμένει (sleep), και η διεργασία V στην κατάσταση όπου

ενεργοποιείται(wake-up). Όταν μια διεργασία επικοινωνήσει με την διεργασία P και το mutex έχει τιμή 0 (false), τότε η διεργασία περιμένει (sleeps), μέχρι κάποια άλλη διεργασία εκτελέσει την V, που σημαίνει πως έχει τελειώσει με το κρίσιμο κομμάτι της (αλλάζει την τιμή του mutex σε 1(true)), και έτσι η διεργασία που περίμενε μπορεί να μπει στο δικό της κρίσιμο σημείο. Με αυτό τον τρόπο αποκλείεται να μπουν και οι δυο διεργασίες να εκτελέσουν το κρίσιμο τους κομμάτι την ίδια χρονική στιγμή. Για να το επαληθεύσουμε αυτό, έχουμε την διεργασία mut_exl, η οποία περιέχει την εντολή assert(!(csp1&&csp2)) που εγγυείται ότι οι μεταβλητές csp1 και csp2 που βρίσκονται στα κρίσιμα σημεία των διεργασιών Process1 και Process2 αντίστοιχα, δεν θα πάρουν ποτέ την τιμή 1 ταυτόχρονα.

```
/*----- Mutual Exclusion using Semaphores -----*/

bool mutex = true;
mtype = {one,two};            //symbolikes times-opws to 'enum' stin C
chan p = [1] of {mtype};      //Dilwsi kanaliou megethous 1
chan v = [1] of {mtype};
int csp1;
int csp2;



active proctype P(){
      do
      ::
            do
            ::    atomic{mutex == false -> skip;}
            ::    atomic{mutex == true -> break;}
            od;
            if
            ::    atomic{
                        v?one;      //to v perimenei tin timi one
                        p!one;      //to p stelnei tin timi one
                        mutex = false;
                  }
            ::    atomic{
                        v?two;
                        p!two;
                        mutex = false;
                  }
            fi;
      od;

}
```

```
active proctype V(){
      do
      //to atomic dilwnei pws oi entoles pou akolouthoun 8a ektelestoun
      //synexomena
      ::    atomic{
                  mutex = true;
                  v!one;

            }
      ::    atomic{
                  mutex = true;
                  v!two;
            }
      od;
}


active proctype Process1(){
progress1:  do
            ::    atomic{
                  p?one;
                  csp1 = 1;
                  printf("Process 1 executes its critical section\n");
                  csp1 = 0;
            }
            printf("Process 1 executes its remainder\n");
            od;
}

active proctype Process2(){
progress2:  do
            ::    atomic{
                  p?two;
                  csp2 = 1;
                  printf("Process 2 executes its critical section\n");
                  csp2 = 0;
            }
            printf("Process 2 executes its remainder\n");
            od;
}


active proctype mut_exl(){
      do
      ::    assert(!(csp1&&csp2));
      od;
}
```

Παράδειγμα 2.1 *Πρόγραμμα αμοιβαίου αποκλεισμού με semaphores*

Χρησιμοποιώντας το εργαλείο XSPIN, έγινε η επαλήθευση του πιο πάνω προγράμματος για τον εντοπισμό πιθανού αδιεξόδου (deadlock) ή και παραβίαση της συνθήκης assert, που θα συνεπαγόταν ότι ο αμοιβαίος αποκλεισμός δεν ισχύει στο πιο πάνω πρόγραμμα. Όπως φαίνεται από το αποτέλεσμα της επαλήθευσης του παραδείγματος (Παράδειγμα 2.2) που ακολουθεί, δεν εντοπίστηκε οποιοδήποτε λάθος, άρα ο κώδικας αμοιβαίου αποκλεισμού του

παραδείγματος είναι ορθός (**errors 0**). Αυτό οφείλεται στο γεγονός ότι οι διεργασίες P και V συγχρονίζουν με σωστό τρόπο το κρίσιμο σημείο των διεργασιών Process1 και Process2.

```
//  Verification result for mutual exclusion (assertions) AND
//  Deadlocks(invalid end states)
(Spin Version 5.1.6 -- 9 May 2008)
      + Partial Order Reduction

Full statespace search for:
      never claim             - (not selected)
      assertion violations    +
      cycle checks            - (disabled by -DSAFETY)
      invalid end states      +

State-vector 48 byte, depth reached 47, errors: 0
      164 states, stored
      372 states, matched
      536 transitions (= stored+matched)
      184 atomic steps
hash conflicts:         0 (resolved)

    2.501   memory usage (Mbyte)

pan: elapsed time 0 seconds
0.00user 0.00system 0:00.00elapsed 71%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+752minor)pagefaults 0swaps
```

Παράδειγμα 2.2 Αποτέλεσμα επαλήθευσης του παραδείγματος για τυχών deadlocks ή και παραβίαση της συνθήκης assert() – **errors: 0**

Εκτός από το πιο πάνω παράδειγμα αμοιβαίου αποκλεισμού με σηματοφόρους, υλοποιήθηκαν και κάποιες άλλες παραλλαγές αμοιβαίου αποκλεισμού. Το πρώτο παράδειγμα αφορά αμοιβαίο αποκλεισμό με exchange instructions. Κάθε διεργασία Pi θέλει να εκτελέσει το κρίσιμο κομμάτι της. Χρησιμοποιούμε μια τοπική μεταβλητή xi για κάθε διεργασία και μια κοινή μεταβλητή bolt που έχει αρχική τιμή 0. Η ανταλλαγή της τιμής της κοινής μεταβλητής bolt με την τιμή της τοπικής μεταβλητής xi, κάνει την τιμή της bolt ίση με 1, απαγορεύοντας την πρόσβαση σε οποιαδήποτε άλλη διεργασία εκτός από την διεργασία i, η οποία εκτελεί το κρίσιμο της κομμάτι και ακολούθως ανταλλάζει και πάλι την τιμή του xi με την bolt δίνοντας την ευκαιρία σε κάποια άλλη διεργασία να εκτελέσει το δικό της κρίσιμο κομμάτι με τον ίδιο τρόπο.

```
Process1(){
x1:=1
        while TRUE do
                x1:=:bolt       //Exchange instruction
                while x1==1 do
                        x1:=: bolt      //Exchange instruction
```

```
            od;
            crit1;
            x1:=:bolt; rem;          //Exchange instruction
        od;}
```

Η δεύτερη υλοποίηση είναι με test and set instructions. Σε αυτή την παραλλαγή, εξασφαλίζουμε ότι αν δυο διεργασίες προσπαθήσουν να διαβάσουν την κοινή μεταβλητή bolt την ίδια στιγμή, αυτή που θα φτάσει και θα διαβάσει την μεταβλητή πρώτη, αλλάζει την τιμή της μεταβλητής σε 1, πριν να επιτραπεί σε άλλη διεργασία να την διαβάσει, εξασφαλίζοντας πως μόνο αυτή εκτελεί το κρίσιμο κομμάτι της.

```
Process1(){
While TRUE do
        x1:=bolt;
        bolt=1;
        while x1==1 do
                x1:=bolt;
                bolt:=1;
        od;
        crit1;
        bolt:=0;
        rem1
}
```

Η τελευταία υλοποίηση είναι με lock instructions. Σε αυτή την περίπτωση η μεταβλητή bolt αρχικοποιείται με 0, και όποια διεργασία την διαβάσει πρώτη της αναθέτει την τιμή 1 "κλειδώνοντας" έτσι την μεταβλητή ώστε να μην μπορεί άλλη διεργασία να βγει από το δεύτερο while και να μπει και αυτή στο κρίσιμο της κομμάτι. Μόλις η διεργασία τελειώσει το κρίσιμο κομμάτι της, κάνει την bolt = 0, έτσι ώστε μόλις κάποια άλλη διεργασία την διαβάσει, να μπει στο κρίσιμο της κομμάτι και να κάνει την ίδια διαδικασία.

```
Process1(){
While TRUE do
        While bolt == 1 do od;
        bolt:=1;
        crit1;
        bolt:=0;
        rem1;
od;
}
```

Οι πιο πάνω παραλλαγές υλοποιήθηκαν και επαληθεύτηκαν στο XSPIN για παραβίαση του αμοιβαίου αποκλεισμού χωρίς λάθη. (Στο παράρτημα Α βρίσκονται οι κώδικες υλοποίησης σε Promela όλων των αλγορίθμων αμοιβαίου αποκλεισμού που περιγράφτηκαν πιο πάνω)

2.7 Χρονική Λογική

Όπως έχουμε προαναφέρει, η επαλήθευση διάφορων ιδιοτήτων στα μοντέλα μπορεί να γίνει με την εντολή assert. Αυτή η εντολή όμως δεν είναι αρκετή, γιατί δεν είναι σε θέση να καλύψει ιδιότητες συστημάτων που αφορούν συμπεριφορά που μεταβάλλεται στον χρόνο. Για παράδειγμα αν θέλουμε να εκφράσουμε την ιδιότητα ότι αν κάποια διεργασία προσπαθήσει σε κάποια στιγμή να εκτελέσει το κρίσιμο της κομμάτι, τότε κάποτε θα το κάνει. Για τον έλεγχο τέτοιων ιδιοτήτων, το XSPIN υποστηρίζει τη γραμμική χρονική λογική LTL (Linear Temporal Logic) [1]. Η LTL παρέχει όλους τους τελεστές του προτασιακού λογισμού, αλλά και κάποιους επιπλέον που αφορούν το χρόνο.

Μια ιδιότητα γραμμένη σε LTL, αποτελείται από ατομικές προτάσεις (p,q), τελεστές του προτασιακού λογισμού, και τελεστές χρονικής λογικής που φαίνονται στον Πίνακα 2.1. Οι ατομικές προτάσεις είναι λογικές εκφράσεις που χρησιμοποιούν μεταβλητές, σταθερές και κατηγορηματικά σύμβολα(<,<=,== κλπ.).

Τελεστής	XSPIN	Εξήγηση
always	[] p	Το p θα συμβαίνει σε κάθε μελλοντική στιγμή
eventually	<> p	Το p θα συμβεί σε κάποια μελλοντική στιγμή
until	p U q	Το p πρέπει να είναι αληθές μέχρι να συμβεί το q

Πίνακας 2.1 *Τελεστές χρονικής λογικής και ο τρόπος αναγνώρισης τους στο XSPIN*

Η χρονική λογική είναι ένα πολύ βοηθητική στην επαλήθευση συστημάτων, γιατί μας βοηθά να εκφράσουμε σημαντικές ιδιότητες [1] όπως:

- ✓ Ασφαλείας: τίποτε κακό δε θα συμβεί, [] !bad
- ✓ Βιωσιμότητας: κάτι καλό τελικά θα συμβεί, <>good
- ✓ Ανταπόκρισης: κάθε αίτηση θα εξυπηρετηθεί, [](request → <> Serviced)
- ✓ Αντιδραστικότητας: ([]<>Enabled)→([]<> executed)

Ένα καλό παράδειγμα για την καλύτερη κατανόηση των τελεστών της LTL και της εφαρμογής τους, είναι η μετατροπή κάποιων ιδιοτήτων των φώτων τροχαίας σε ιδιότητες χρονικής λογικής [1].

- Από κόκκινο το φώς δεν μπορεί να γίνει αμέσως πράσινο
  [](κόκκινο→!<> πράσινο)
- Στο μέλλον το φώς θα ξαναγίνει πράσινο
  <> πράσινο
- Από κόκκινο το φώς γίνεται πράσινο αφού περάσει από το κίτρινο για κάποιο χρονικό διάστημα.
  [](κόκκινο→((κόκκινο U κίτρινο) U πράσινο))

Στο Παράδειγμα 2.1 μπορούμε να ελέγξουμε αν ο κώδικας πάσχει από λιμό (starvation), δηλαδή αν κάποια διεργασία προσπαθήσει σε κάποια στιγμή να εκτελέσει το κρίσιμο της κομμάτι, τότε όντως κάποτε θα το κάνει, και δεν θα περιμένει για πάντα. Για να το ελέγξουμε αυτό χρησιμοποιούμε τον LTL manager που παρέχει το XSPIN και δίνουμε την ιδιότητα ([]<>p)&&([]<>q), όπου τα p και q δηλώνουν ότι οι διεργασίες Process1 και Process2 βρίσκονται στο κρίσιμο κομμάτι τους. Η ιδιότητα επαληθεύει ότι και οι δυο διεργασίες πάντοτε στο μέλλον θα εκτελούν το κρίσιμο κομμάτι τους(δεν υπάρχει starvation), και είναι ορθή όπως φαίνεται και από το αποτέλεσμα στο Παράδειγμα 2.3.

```
#define p   (csp1 == 1)
#define q   (csp2 == 1)

      /*
       * Formula As Typed:
       */

never {    /* (([]  <>  p)&&([]  <>q)  ) */
T0_init:
      if
      :: ((p) && (q)) -> goto accept_S81
      :: ((p)) -> goto T1_S81
      :: (1) -> goto T0_init
      fi;
accept_S81:
      if
      :: (1) -> goto T0_init
      fi;
T1_S81:
      if
      :: ((q)) -> goto accept_S81
      :: (1) -> goto T1_S81
      fi;
}
#ifdef NOTES
#endif
#ifdef RESULT
warning: for p.o. reduction to be valid the never claim must be stutter-
invariant
(never claims generated from LTL formulae are stutter-invariant)
depth 0: Claim reached state 7 (line 99)
```

```
depth 0: Claim reached state 7 (line 100)
(Spin Version 5.1.6 -- 9 May 2008)
      + Partial Order Reduction
Full statespace search for:
      never claim             +
      assertion violations    + (if within scope of claim)
      acceptance   cycles     + (fairness enabled)
      invalid end states      - (disabled by never claim)

State-vector 52 byte, depth reached 88, errors: 0
      164 states, stored
      372 states, matched
      536 transitions (= stored+matched)
      184 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    0.011   equivalent memory usage for states (stored*(State-vector +
overhead))
    0.285   actual memory usage for states (unsuccessful compression:
2682.28%)
            state-vector as stored = 1808 byte + 16 byte overhead
    2.000   memory used for hash table (-w19)
    0.305   memory used for DFS stack (-m10000)
    2.501   total actual memory usage

pan: elapsed time 0 seconds
0.00user 0.00system 0:00.00elapsed 50%CPU (0avgtext+0avgdata 0maxresident)k
0inputs+0outputs (0major+755minor)pagefaults 0swaps

#endif
```

Παράδειγμα 2.3 Αποτέλεσμα επαλήθευσης του παραδείγματος για starvation – **errors: 0**

Κεφάλαιο 3

Πολυπύρηνοι Επεξεργαστές

3.1 Εισαγωγή στους Πολυπύρηνους Επεξεργαστές

Ένας πολυπύρηνος επεξεργαστής είναι ένας επεξεργαστής ο οποίος αποτελείται από δύο ή περισσότερους ανεξάρτητους πυρήνες, πάνω στους οποίους υπάρχει η δυνατότητα να τρέχουν πολλές διεργασίες ταυτόχρονα. Αυτό γίνεται κατορθωτό αφού οι διεργασίες που τρέχουν σε ανεξάρτητους πυρήνες μπορούν να επικοινωνούν μεταξύ τους όταν αυτό χρειάζεται. Σε συστήματα κοινής μνήμης (shared memory), οι πυρήνες διασυνδέονται μεταξύ τους χρησιμοποιώντας κάποια κοινή μνήμη μέσω της οποίας επικοινωνούν και οι διεργασίες για ανταλλαγή πληροφοριών κάνοντας χρήση κοινών μεταβλητών. Αντιθέτως σε συστήματα κατανεμημένης μνήμης (distributed memory), κάθε επεξεργαστής έχει την δική του μνήμη, και η επικοινωνία μεταξύ των πυρήνων και των διεργασιών γίνεται με ανταλλαγή – διαβίβαση μηνυμάτων (message passing) [4].

3.2 Στάδια εξέλιξης από μονοπύρηνους σε πολυπύρηνους επεξεργαστές

Ένα εύλογο ερώτημα είναι το πώς οδηγηθήκαμε σε πολυπύρηνους (multi-core) επεξεργαστές και δεν συνεχίσαμε να αυξάνουμε την λειτουργική συχνότητα (operating frequency) ενός μονοπύρηνου (single-core) επεξεργαστή. Η απάντηση σε αυτό το ερώτημα

είναι το γεγονός ότι η αύξηση της λειτουργικής συχνότητας δεν απέδιδε την ανάλογη αύξηση στην απόδοση του επεξεργαστή. Αυτό οφείλεται κυρίως σε τρεις παράγοντες [10,11]:

- *Memory wall:* Η αυξανόμενη διαφορά ανάμεσα στις ταχύτητες του επεξεργαστή και τις μνήμης. Το κόστος ανάκτησης δεδομένων από την μνήμη (memory latency) αυξανόταν συνεχώς και για να γίνει κάποιος συμβιβασμός αυξάνονταν και τα μεγέθη των μνήμων cache. Μετά όμως από κάποιο σημείο, το μέγεθος της cache μνήμης ήταν πολύ μεγάλο και δεν εξυπηρετούσε τον σκοπό της.

- *ILP (Instruction Level Parallelism):* Η αυξανόμενη δυσκολία εύρεσης κάποιου συνδυασμού για παραλληλισμό εντολών σε ικανοποιητικό επίπεδο για να κρατείτε ένας μονοπύρηνος επεξεργαστής συνεχώς απασχολημένος. Το ILP ήταν μια τεχνική μεγιστοποίησης της απόδοσης του επεξεργαστή. Στόχος της ήταν να αυξήσει τον αριθμό εντολών που εκτελούνται από τον επεξεργαστή σε κάθε clock cycle.

- *Power Wall*: Η συνεχής αύξηση της λειτουργικής συχνότητας ενός μονοπύρηνου επεξεργαστή έχει ως άμεσο αποτέλεσμα την εκθετική αύξηση κατανάλωσης ενέργειας, καθώς επίσης και της θερμότητας του επεξεργαστή σε σημείο στο οποίο το υλικό δεν άντεχε πλέον τις τόσο ψηλές θερμοκρασίες.

Ακόμα ένας λόγος που οδήγησε στην ανάπτυξη των πολυπύρηνων ήταν το γεγονός ότι σε ένα μονοπύρηνο δεν υπήρχε η δυνατότητα για παράλληλη επεξεργασία. Υπήρχε μόνο η έννοια του κατανεμημένου υπολογισμού (distributed computing), δηλαδή η ταυτόχρονη επεξεργασία μεγάλων όγκων δεδομένων από μια ομάδα ξεχωριστών μονοπύρηνων ηλεκτρονικών υπολογιστών, που ήταν ενωμένα σε κάποιο δίκτυο και επικοινωνούσαν μεταξύ τους μέσω ανταλλαγής μηνυμάτων, αλλά αυτό δεν ήταν αρκετό. Η συνεχείς και ραγδαία αύξηση των αποθηκευμένων δεδομένων στις διάφορες βάσεις δεδομένων, αλλά και οι σύνθετοι τρόποι αποθήκευσης τους για εξοικονόμηση χώρου στους διάφορους εξυπηρετητές, έφεραν την επιτακτική ανάγκη για αύξηση στην δύναμη επεξεργασίας του επεξεργαστή ενός υπολογιστή. Επιπλέον, με την ανάπτυξη και εξέλιξη του τρόπου σχεδίασης και υλοποίησης των διάφορων λογισμικών, εμφανίστηκε η δυνατότητα να τρέχουν πολλαπλές διεργασίες ταυτόχρονα σε κάποιες εφαρμογές. Ως αποτέλεσμα των πιο πάνω ήταν η ανάγκη που δημιουργήθηκε για αύξηση της υπολογιστικής δύναμης των επεξεργαστών, χρησιμοποιώντας παράλληλη επεξεργασία δεδομένων.

Για την υποστήριξη αυτής της δυνατότητας, υιοθετήθηκαν αρχικά κάποιες προσεγγίσεις όπως για παράδειγμα η χρήση λειτουργικών συστημάτων που επέτρεπαν την ταυτόχρονη εκτέλεση πολλών διεργασιών (multitasking). Στην ουσία όμως αυτές οι προσεγγίσεις το μόνο που πέτυχαν ήταν "κρύψουν" κατά κάποιο τρόπο το χρόνο ανάκτησης δεδομένων από την μνήμη, δίνοντας την άδεια σε κάποια άλλη διεργασία να χρησιμοποιήσει τον επεξεργαστή, μέχρι να ανακτηθούν από την μνήμη τα δεδομένα που χρειαζόταν η προηγούμενη διεργασία. Δεν εξυπηρετούσε δηλαδή τον αρχικό στόχο που ήταν η παράλληλη επεξεργασία.

Μια άλλη τεχνική που χρησιμοποιήθηκε και η οποία επέτρεπε την παράλληλη επεξεργασία, ήταν ή αύξηση των επεξεργαστών σε κάθε υπολογιστή (multiprocessor systems). Αυτά τα συστήματα όμως είχαν το μεγάλο μειονέκτημα του υψηλού κόστους παραγωγής. Το αμέσως επόμενο βήμα, ήταν η έρευνα που ξεκίνησε για την αποδοτικότητα χρήσης πολλών πυρήνων σε έναν επεξεργαστή. Αφού τα ποσοστά αποδοτικότητας αυτής της αρχιτεκτονικής ήταν άκρως ικανοποιητικά, οι πολυπύρηνοι επεξεργαστές θεωρήθηκαν ως το μέλλον στο πεδίο σχεδίασης επεξεργαστών. Για τους πιο πάνω λόγους, μεγάλες εταιρείες κατασκευής επεξεργαστών όπως η Intel και η AMD, έστρεψαν την προσοχή τους στους πολυπύρηνους επεξεργαστές, θυσιάζοντας τα χαμηλά κόστη παραγωγής (ενός μονοπύρηνου επεξεργαστή) για υψηλότερη απόδοση.

Η Intel [10] προσπάθησε μέσω μιας έρευνας (Tera-scale Computing Research Program) να βρει τρόπους να προσαρμόσει τις υπολογιστικές δυνατότητες ενός υπέρ-υπολογιστή σε συσκευές καθημερινής χρήσης όπως desktops, laptops κλπ. Στην έρευνα αυτή χρησιμοποιούνται εκατοντάδες επεξεργαστές οι οποίοι επεξεργάζονται τεράστιους όγκους δεδομένων παράλληλα. Η Intel στήριξε την ανάπτυξη πολυπύρηνων επεξεργαστών στους εξής παράγοντες:

- Απόδοση: Μέσω της παράλληλης επεξεργασίας σε ένα πολυπύρηνο επεξεργαστή θα γίνει κατορθωτή η αύξηση της απόδοσης, αφού με τον κατάλληλο συγχρονισμό των πυρήνων θα μειωθεί ο χρόνος υπολογισμού σε σχέση με τους μονοπύρηνους επεξεργαστές.

- Κατανάλωση Ενέργειας: Η σκέψη της Intel είναι να χρησιμοποιήσει μια τεχνική σύμφωνα με την οποία οι πυρήνες που δεν είναι απασχολημένοι θα "σβήνουν" για μείωση της κατανάλωσης ενέργειας σε περιόδους που αυτοί είναι αδρανής.

Σε αυτή την έρευνα της Intel εντοπίστηκαν κάποιες προκλήσεις σε δυο ερευνητικά πεδία:

1. Όσον αφορά την αρχιτεκτονική των μικροεπεξεργαστών έπρεπε να ληφθούν υπόψη τέσσερις παράγοντες :

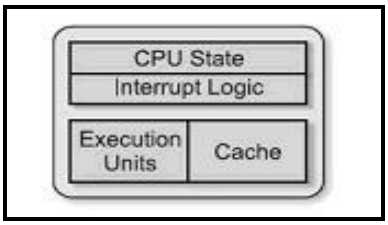
   I. Βελτιστοποιημένος σχεδιασμός των πυρήνων: Το multithreading μπορεί να βοηθήσει στη μείωση του χρόνου ανάκτησης δεδομένων από την μνήμη και στη μείωση της περιττής πολυπλοκότητας

   II. Δημιουργία μονάδων που θα εκτελούν προκαθορισμένες εργασίες π.χ. επιταχυντές δικτύου(network accelerators) ή μηχανές κρυπτογράφησης(cryptography engines). Αυτές οι μονάδες θα αποδίδουν καλύτερα στις συγκεκριμένες εργασίες από κάποια μονάδα γενικής λειτουργίας.

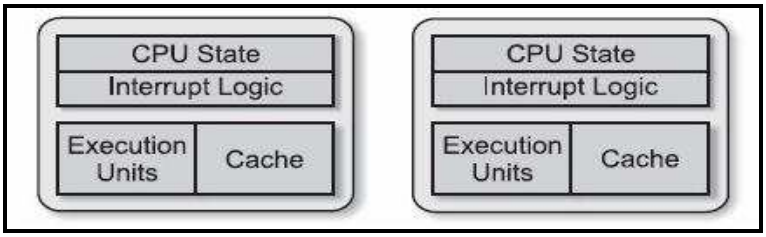
   III. Scalable On-die Interconnect fabric: Ένας επεξεργαστής χρειάζεται να συνδέεται όχι μόνο με ένα μεγάλο αριθμό πυρήνων αλλά και με μνήμες cache και με εξειδικευμένο υλικό π.χ. κάρτες γραφικών. Συνεπώς η σύνδεση αυτή πρέπει να είναι εξελίξιμη, δηλαδή να μπορεί να υποστηρίζει μεταβλητό αριθμό πυρήνων, μνήμων κλπ.

   IV. Σχεδιασμός κυκλωμάτων με ενσωματωμένη μνήμη έτσι ώστε να γίνετε πιο αποδοτική χρήση της ενέργειας καθώς επίσης και με δυνατότητα ρύθμισης της αποδοτικότητας της ενέργειας. Δηλαδή εργασίες οι οποίες χαρακτηρίζονται ως μη κρίσιμες θα μπορούσαν να τρέχουν σε χαμηλότερες συχνότητες για μείωση της κατανάλωσης της ενέργειας.

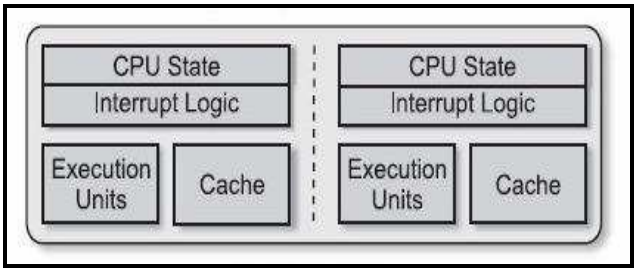
2. Στη σχεδίαση και ανάπτυξη νέων εφαρμογών θα πρέπει να λαμβάνεται υπόψη το γεγονός ότι θα τρέχουν σε πολυπύρηνο επεξεργαστή έτσι ώστε να γίνεται εκμετάλλευση της παράλληλης επεξεργασίας για αύξηση της επίδοσης. Αυτό όμως δεν είναι εύκολο, γιατί οι προγραμματιστές θα χρειάζονται παραπάνω χρόνο για κωδικοποίηση και αποσφαλμάτωση τέτοιων προγραμμάτων. Ο λόγος είναι η δυσκολία που παρουσιάζεται στην εφαρμογή σωστού συγχρονισμού ανάμεσα στα διάφορα threads χωρίς να παρουσιάζονται προβλήματα όπως π.χ. Deadlocks και starvation.



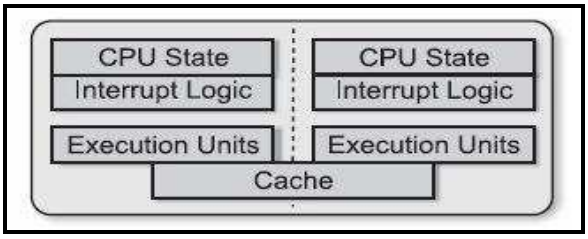
(A) Single-core



(B) Multiprocessor System



(C) Multi-core



(D) Multi-core with shared cache

Εικόνα 3.1: *Γραφική απεικόνιση αρχιτεκτονικής Single-core, Multiprocessor System, Multi-core και Multi-core με κοινή cache*

3.3 Πλεονεκτήματα Πολυπύρηνων Επεξεργαστών

1. Με την χρήση πολλών πυρήνων θα υπάρξει και η ανάλογη αύξηση στην απόδοση αφού οι υπολογισμοί θα γίνονται παράλληλα και συνεπώς θα μειωθεί ο χρόνος εκτέλεσης κάποιου προγράμματος ή κάποιου υπολογισμού. Η αύξηση στην απόδοση όμως εξαρτάται και από το πόσο καλά είναι γραμμένο το πρόγραμμα για να τρέχει σε πολλούς πυρήνες.

2. Επίσης υπάρχει η δυνατότητα εκτέλεσης πολλών προγραμμάτων ταυτόχρονα ορίζοντας ένα πυρήνα για το κάθε πρόγραμμα, αποφεύγοντας την όποια καθυστέρηση θα είχαμε σε μονοπύρηνο επεξεργαστή όπου κάθε πρόγραμμα θα περίμενε την σειρά του για να πάρει τον έλεγχο του μοναδικού επεξεργαστή.

3. Σε σχέση με τα συστήματα με πολλούς επεξεργαστές (multiprocessor), ένας πολυπύρηνος επεξεργαστής με X πυρήνες καταναλώνει λιγότερη ενέργεια από ένα σύστημα που υποστηρίζει X μονοπύρηνους επεξεργαστές, λόγω του ότι τα σήματα ταξιδεύουν μέσα στο ίδιο chip και δεν χρειάζεται να ταξιδεύουν μεγάλες αποστάσεις.

4. Η δυνατότητα να έχουμε πολλούς πυρήνες στο ίδιο chip, επιτρέπει στα σήματα τα οποία στέλνονται μεταξύ των πυρήνων να ταξιδεύουν μικρότερες αποστάσεις και ως αποτέλεσμα να υποβιβάζονται λιγότερο. Αυτά τα υψηλότερης ποιότητας σήματα επιτρέπουν την μεταφορά περισσότερων δεδομένων σε κάποια χρονική περίοδο, αφού είναι συντομότερα και δεν χρειάζεται να επαναλαμβάνονται συχνά.

5. Σε περίπτωση βλάβης κάποιου πυρήνα μπορούμε απλά να τον βγάλουμε εκτός λειτουργίας, χωρίς να μειωθεί δραματικά η απόδοση του επεξεργαστή. Η ιδιότητα αυτή είναι γνωστή και ως ανοχή σφαλμάτων (Fault-tolerance).

3.4 Μειονεκτήματα Πολυπύρηνων Επεξεργαστών

1. Ο προγραμματισμός παράλληλου κώδικα απαιτεί πολύπλοκο συγχρονισμό των διεργασιών και μπορεί εύκολα να παρουσιάσει μικρά και δυσεύρετα λάθη λόγω της παρεμβολής των διεργασιών σε δεδομένα που βρίσκονται σε κοινή μνήμη. Τέτοιος κώδικας είναι πολύ δύσκολο να ελεγχθεί και να αποσφαλματωθεί. Επίσης οι προγραμματιστές θα πρέπει να ξεφύγουν από τον παραδοσιακό τρόπο σκέψης για σχεδίαση και υλοποίηση προγραμμάτων, και να ακολουθήσουν νέες τεχνικές οι οποίες υποστηρίζουν τη σωστή και εύκολη παραγωγή παράλληλου κώδικα που να μπορεί να τρέχει σε πολλούς πυρήνες.

2. Οι ψηλές θερμοκρασίες που αναγκαστικά θα υπάρχουν λόγω του ότι θα συνυπάρχουν πολλοί πυρήνες σε έναν επεξεργαστή συνεπάγεται μικρότερη διάρκεια ζωής του επεξεργαστή. Επίσης οι ψηλές θερμοκρασίες είναι αποτέλεσμα της ενέργειας που καταναλώνεται για να λειτουργούν οι πυρήνες. Είναι δεδομένο πως ένας πολυπύρηνος επεξεργαστής καταναλώνει περισσότερη ενέργεια από οποιοδήποτε μονοπύρηνο επεξεργαστή, αλλά αυτό είναι το αντίτιμο της ψηλότερης απόδοσης. Προς το παρών το πρόβλημα ψηλών θερμοκρασιών στους μονοπύρηνους επεξεργαστές αντιμετωπίζεται διατηρώντας τις λειτουργικές τους συχνότητες σε χαμηλά επίπεδα.

3. Για να λειτουργεί ένας γρήγορος πολυπύρηνος επεξεργαστής χρειάζεται μεγάλες ποσότητες ηλεκτρικής ενέργειας γεγονός που κάνει το κύκλωμα επιρρεπή σε ηλεκτρικό θόρυβο, ο οποίος δημιουργεί παρεμβάσεις. Επειδή τα μονοπάτια πάνω σε ένα κύκλωμα είναι πολύ κοντά το ένα στο άλλο, όσο περισσότερη ενέργεια τρέχει μέσα στα μονοπάτια, τόσο περισσότερα λάθη γίνονται στα δεδομένα λόγω της ηλεκτρικής ακτινοβολίας που εκπέμπεται.

4. Η αύξηση του αριθμού των πυρήνων σε κάποιον επεξεργαστή δεν οδηγεί πάντα σε καλύτερες επιδόσεις. Δεν είναι απόλυτο δηλαδή ότι με το να ενσωματώσουμε περισσότερους πυρήνες σε κάποιον επεξεργαστή θα έχουμε και την ανάλογη αύξηση στην απόδοση.

3.5 Προκλήσεις πολυπύρηνων επεξεργαστών και παράλληλης επεξεργασίας

Μια από τις πιο δυνατές προκλήσεις αγγίζει τον τομέα σχεδίασης και δημιουργίας προγραμμάτων τα οποία να επωφελούνται από τα πλεονεκτήματα που προσφέρουν οι πολυπύρηνοι επεξεργαστές και ο παραλληλισμός. Με την εισαγωγή των πολυπύρηνων επεξεργαστών επήλθε και η ανάγκη μετατροπής αλγορίθμων και προγραμμάτων ώστε να μπορούν να τρέχουν παράλληλα, και να εκμεταλλευτούν την ύπαρξη πολλών πυρήνων στο ίδιο chip. Αυτό απαιτεί από τους προγραμματιστές να σχεδιάζουν και να υλοποιούν παράλληλα προγράμματα, διαδικασία η οποία είναι επίπονη, χρονοβόρα και επιρρεπής σε λάθη. Η παραγωγή τέτοιου κώδικα προβλέπεται ότι θα είναι ο μεγαλύτερος περιορισμός στην πλήρη εκμετάλλευση των δυνατοτήτων των πολυπύρηνων επεξεργαστών[10].

Παρόλα αυτά υπάρχουν κάποιες τεχνικές με τις οποίες μπορούν να μετατραπούν κάποιοι σειριακοί αλγόριθμοι σε παράλληλους. Μια συνηθισμένη τεχνική που χρησιμοποιείται συχνά έχει τα ακόλουθα βήματα:

I. *Αποσύνθεση (Decomposition):* Σε πρώτο βήμα γίνεται η αναγνώριση κομματιών του προβλήματος τα οποία μπορούν να εκτελεστούν παράλληλα και χωρίζονται σε υπο-προβλήματα.

II. Χαρτο*γράφηση (Mapping):* Σε αυτό το βήμα γίνεται αντιστοίχηση των υπο-προβλημάτων που δημιουργήθηκαν στην προηγούμενη φάση στα διαθέσιμα threads.

III. *Έλεγχος πρόσβασης (Access control):* Τέλος γίνεται ο συγχρονισμός και ο έλεγχος πρόσβασης σε κοινά δεδομένα από πολλαπλά threads.

Μια άλλη πρόκληση είναι οι βασικές ιδιότητες που πρέπει να πληρούν όλα τα παράλληλα προγράμματα, δηλαδή ιδιότητες ασφαλείας οι οποίες εγγυώνται ότι τίποτα κακό δε θα συμβεί ποτέ π.χ. αδιέξοδα (Deadlock), και ιδιότητες βιωσιμότητας οι οποίες εγγυώνται ότι ένα πρόγραμμα τελικά θα κάνει πρόοδο π.χ. ότι κάθε πρόγραμμα κάποτε τελειώνει.

Όσον αφορά την μνήμη, η μεγαλύτερη πρόκληση βρίσκεται στο σχεδιασμό διάφορων αποδοτικών μηχανισμών που θα εξασφαλίσουν την συνέπεια ανάμεσα στα διάφορα επίπεδα

των μνήμων cache. Αφού ο κάθε πυρήνας έχει την δική του μνήμη cache στην οποία βρίσκεται ένα αντίγραφο των δεδομένων που έχουν και οι άλλοι πυρήνες, είναι απαραίτητο αυτό το αντίγραφο να είναι πάντοτε το πιο ενημερωμένο. Όταν δηλαδή ένας πυρήνας κάνει κάποιες αλλαγές στα δεδομένα της μνήμης cache ή της κύριας μνήμης, αυτές οι αλλαγές πρέπει να υλοποιούνται αμέσως και στις μνήμες των άλλων πυρήνων. Έτσι, όταν κάποιος άλλος πυρήνας προσπαθήσει να διαβάσει τα δεδομένα αυτά, ή να τα τροποποιήσει, θα εκτελέσει την λειτουργία στα πιο ενημερωμένα δεδομένα, και ως αποτέλεσμα θα διατηρούνται ορθοί οι υπολογισμοί που γίνονται κατά την παράλληλη επεξεργασία.

Άλλες προκλήσεις αφορούν τα διάφορα κόστη που παρουσιάζονται με τη χρήση πολλών πυρήνων και παράλληλου προγραμματισμού. Για παράδειγμα το κόστος που χρειάζεται για την εκκίνηση/τερματισμό παράλληλων διεργασιών και αποστολή μηνυμάτων μεταξύ τους. Ακόμα το κόστος συγχρονισμού περιορίζει την επίδοση γιατί στην πραγματικότητα μετατρέπει σε σειριακό κάποιο κομμάτι του προγράμματος και έτσι περιορίζει τον αριθμό των παράλληλων διεργασιών. Επίσης ο ίσος καταμερισμός φορτίου (load balancing) είναι μια δυνατή πρόκληση για τον προγραμματιστή. Ο διαμοιρασμός δηλαδή του φόρτου εργασίας πρέπει να γίνεται εξίσου ανάμεσα στους πυρήνες έτσι ώστε ο ο χρόνος που παραμένει αδρανής ένας πυρήνας να κρατείται στο ελάχιστο και να μειώνεται ο χρόνος υπολογισμού, αυξάνοντας την απόδοση. Όλα τα πιο πάνω κόστη είναι αναπόφευκτα και πρέπει να διατηρούνται στο ελάχιστο.

Κεφάλαιο 4

Συνοχή Μνήμης

4.1 Συνοχή μνήμης cache σε πολυπύρηνους επεξεργαστές

Η cache είναι ένα μικρό και γρήγορο κομμάτι μνήμης που ενσωματώνεται στους επεξεργαστές, με σκοπό να αποθηκεύονται αντίγραφα από τα πιο πρόσφατα δεδομένα που ανακτήθηκαν από την κύρια μνήμη. Η μνήμες cache αποτέλεσαν τον βασικότερο παράγοντα αύξησης της απόδοσης και των ταχυτήτων στους μικροεπεξεργαστές. Χρησιμοποιώντας cache μειώνεται ο μέσος όρος του χρόνου που χρειάζεται ένας επεξεργαστής για πρόσβαση στη μνήμη, αφού πρώτα ελέγχει στην τοπική cache του για τα ζητούμενα δεδομένα και μόνο αν δε βρεθούν εκεί θα ψάξει στην κύρια μνήμη. Τα πράγματα ήταν σχετικά απλά στους μονοπύρηνους επεξεργαστές αφού υπήρχε μόνο ένας επεξεργαστής και μόνο μια cache. Με την δημιουργία όμως των πολυπύρηνων επεξεργαστών, τα πράγματα περιπλέχθηκαν αρκετά.

Το πρόβλημα εστιάζεται στην σωστή διάδοση και αποθήκευση των ορθών δεδομένων στις μνήμες των επεξεργαστών. Για παράδειγμα αν δυο επεξεργαστές P1 και P2 φορτώσουν την ίδια μεταβλητή από την κύρια μνήμη στην τοπική τους μνήμη(cache), και ο P1 αλλάξει την τιμή της, τότε οι δυο επεξεργαστές θα έχουν διαφορετική τιμή για την ίδια μεταβλητή. Ως αποτέλεσμα, ο P2 θα εκτελεί πράξεις πάνω σε λανθασμένα ή παλιά δεδομένα και άρα τα αποτελέσματα του θα είναι επίσης λανθασμένα. Για να είναι ορθό ένα κύκλωμα με δυο ή περισσότερους επεξεργαστές, πρέπει με κάποιο τρόπο ο P1 να ενημερώσει τους υπόλοιπους για την αλλαγή που έκανε πάνω στην μεταβλητή του έτσι ώστε να διατηρηθεί η συνοχή

μνήμης, δηλαδή όλοι οι επεξεργαστές να χρησιμοποιούν πάντα την τελευταία έκδοση των δεδομένων.

Για να έχουμε συνοχή μνήμης πρέπει να τηρούνται οι πιο κάτω κανόνες:

- Όταν υπάρχει εγγραφή ενός κομματιού μνήμης Χ από ένα επεξεργαστή Ρ, η οποία ακολουθείται από μια ανάγνωση στο ίδιο κομμάτι μνήμης Χ από τον ίδιο επεξεργαστή Ρ, χωρίς να υπάρχει οποιαδήποτε άλλη ενδιάμεση εγγραφή από άλλο επεξεργαστή στο κομμάτι Χ, τότε το Χ πρέπει να έχει και να επιστρέφει πάντα την τιμή που έγραψε ο Ρ.
- Όταν υπάρχει ανάγνωση ενός κομματιού μνήμης Χ από ένα επεξεργαστή Ρ1 η οποία γίνεται μετά την εγγραφή στο Χ από τον Ρ2,και δεν υπάρχει οποιαδήποτε άλλη ενδιάμεση εγγραφή από άλλο επεξεργαστή, τότε η τιμή που θα διαβάζει ο Ρ1 πρέπει να είναι η τιμή που έχει γράψει ο Ρ2.
- Εγγραφές στο ίδιο κομμάτι μνήμης πρέπει να γίνονται με σειρά και να επιστρέφεται πάντα η τιμή της τελευταίας εγγραφής. Αν δηλαδή το κομμάτι μνήμης Χ έλαβε δυο τιμές Α και Β με σειρά, από δυο οποιουσδήποτε επεξεργαστές, τότε κανένας επεξεργαστής δεν μπορεί να διαβάσει το Χ σαν Β και μετά Α.

Ένας τρόπος με τον οποίο μπορούμε να επιβάλουμε και να ελέγχουμε την συνοχή της τοπικής μνήμης των επεξεργαστών είναι να συμπεριλάβουμε σε κάθε κομμάτι μνήμης ένα πεδίο, το οποίο θα κωδικοποιεί τα δικαιώματα (permissions) όλων των επεξεργαστών στο συγκεκριμένο κομμάτι. Αν δηλαδή ο επεξεργαστής Ρ1 θέλει να κάνει ανάγνωση ή εγγραφή στη μεταβλητή Χ, πρέπει πρώτα να κοιτάξει στην τοπική του μνήμη και όταν βρει το Χ, να κοιτάξει αν το Χ έχει τα κατάλληλα δικαιώματα για ανάγνωση ή εγγραφή. Αν κάποιος επεξεργαστής επιθυμεί να γράψει σε κάποια μεταβλητή Χ η οποία βρίσκεται και στη μνήμη άλλων επεξεργαστών, τότε πρέπει να ανακτήσει πρώτα δικαίωμα για εγγραφή, αφαιρώντας οποιοδήποτε άλλο δικαίωμα για ανάγνωση έχουν οι υπόλοιποι επεξεργαστές πάνω στην Χ. Γενικά, πρέπει πάντα να ισχύει η πιο κάτω αμετάβλητη συνθήκη για κάθε κομμάτι μνήμης:

”*Σε οποιαδήποτε χρονική στιγμή, τα δικαιώματα για κάποιο κομμάτι μνήμης πρέπει να επιτρέπουν είτε ανάγνωση από πολλούς, είτε εγγραφή από έναν και μόνο επεξεργαστή* ” [15]

Τα δικαιώματα που έχουν τα κομμάτια μνήμης απεικονίζονται με την κατάσταση του κάθε block. Τα περισσότερα πρωτόκολλα συνοχής μνήμης, χρησιμοποιούν τις καταστάσεις που φαίνονται στον Πίνακα 4.1, εξασφαλίζοντας ότι ισχύουν πάντα και οι αντίστοιχες αμετάβλητες συνθήκες.

Κατάσταση	**Δικαιώματα**	**Αμετάβλητη Συνθήκη**
Modified (M)	Ανάγνωση και Εγγραφή	Όλες οι άλλες cache είναι σε I ή NP
Exclusive (E)	Ανάγνωση και Εγγραφή	Όλες οι άλλες cache είναι σε I ή NP
Owned (O)	Ανάγνωση και Εγγραφή	Όλες οι άλλες cache είναι σε S, I ή NP
Shared (S)	Ανάγνωση	Καμιά άλλη cache δεν είναι σε M ή E
Invalid (I)	--	--
Not Present (NP)	--	--

Πίνακας 4.1 *Καταστάσεις, δικαιώματα και αμετάβλητες συνθήκες σε πρωτόκολλα συνοχής μνήμης*

Για παράδειγμα ένας επεξεργαστής μπορεί να γράψει σε κάποιο κομμάτι μνήμης X αν η κατάσταση του κομματιού είναι M ή E, γιατί το πρωτόκολλο εξασφαλίζει ότι όλα τα αντίγραφα του X στις άλλες μνήμες βρίσκονται σε κατάσταση I ή NP. Ένας επεξεργαστής μπορεί να διαβάσει στο X αν η κατάσταση του είναι μια από {M, E, O, S}. Το πρωτόκολλο εξασφαλίζει ότι οι αμετάβλητες συνθήκες ισχύουν πάντοτε, μέσω ανταλλαγής κατάλληλων μηνυμάτων μεταξύ των επεξεργαστών ανάλογα με την ενέργεια που εκτελείται (ανάγνωση ή εγγραφή) και την κατάσταση στην οποία βρίσκεται το συγκεκριμένο κομμάτι μνήμης.

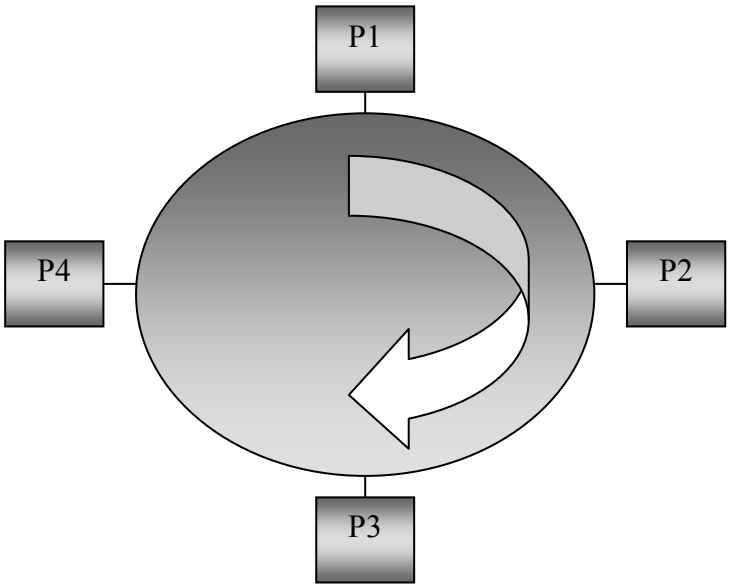
Σε περίπτωση που κάποιος επεξεργαστής δεν βρει την μεταβλητή που χρειάζεται για να διαβάσει (read miss) στην τοπική του μνήμη ή η κατάσταση της είναι I, τότε εκδίδει ένα αίτημα για ανάκτηση της μεταβλητής με δικαίωμα για ανάγνωση. Το πρωτόκολλο είναι υπεύθυνο να βρει την πιο πρόσφατη έκδοση της μεταβλητής, να αφαιρέσει το δικαίωμα εγγραφής από τον επεξεργαστή που το κατέχει (αφήνοντας τον σε κατάσταση με δικαίωμα για ανάγνωση μόνο) και να τον υποχρεώσει να στείλει τη μεταβλητή στον επεξεργαστή που την χρειάζεται. Αν η μεταβλητή δεν υπάρχει σε κάποια cache ή υπάρχει αλλά δεν είναι σε κατάσταση {M, E, O, S} τότε η μεταβλητή ανακτάται από την μνήμη.

Στην αντίστοιχη περίπτωση που ο επεξεργαστής δεν βρει την μεταβλητή που χρειάζεται για να γράψει, τότε εκδίδει ένα αίτημα για ανάκτηση της μεταβλητής με δικαίωμα για εγγραφή. Το πρωτόκολλο ακολουθεί την ίδια διαδικασία που περιγράφτηκε πιο πάνω για ανάγνωση, αλλά εξασφαλίζει επίσης ότι αφαιρούνται και όλα τα δικαιώματα για ανάγνωση από τους υπόλοιπους επεξεργαστές.

4.2 Συνοχή μνήμης σε τοπολογίες δακτυλίου

4.2.1 Τοπολογία δακτυλίου

Η τοπολογία δακτυλίου είναι μια τοπολογία δικτύου, στην οποία όλες οι συσκευές, στην περίπτωση μας επεξεργαστές, είναι ενωμένοι διαδοχικά μεταξύ τους με μορφή ενός κύκλου, έτσι ώστε κάθε επεξεργαστής να συνδέεται με ακριβώς δυο άλλους, έναν σε κάθε πλευρά του. Σε δακτυλίους μονής κατεύθυνσης (Εικόνα 4.1), όλα τα μηνύματα ταξιδεύουν από επεξεργαστή σε επεξεργαστή (point to point) στην ίδια κατεύθυνση, ενώ σε δακτυλίους διπλής κατεύθυνσης τα μηνύματα μπορούν να ταξιδέψουν και στις δυο κατευθύνσεις. Δηλαδή ο κάθε επεξεργαστής μπορεί να στείλει μηνύματα και στον προηγούμενο και στον επόμενο του επεξεργαστή



Εικόνα 4.1 *Επεξεργαστές συνδεδεμένοι σε τοπολογία δακτυλίου μονής κατεύθυνσης*

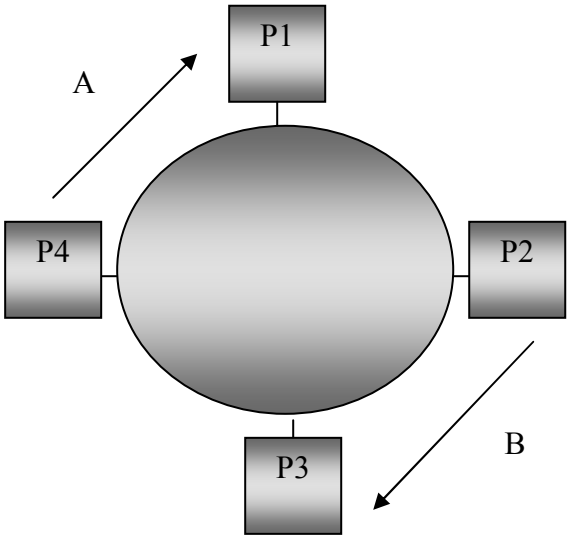
Η τοπολογία δακτυλίου προσφέρει αρκετά πλεονεκτήματα. Είναι απλή αρχιτεκτονική, αποδοτική και προσφέρει γρήγορους συνδέσμους από επεξεργαστή σε επεξεργαστή για την μετάδοση μηνυμάτων, και αν κάποιος επεξεργαστής δεν είναι σε θέση να λάβει κάποιο μήνυμα που στάλθηκε για αυτόν, γιατί μπορεί να επεξεργάζεται κάποιο άλλο μήνυμα, τότε το μήνυμα θα ταξιδεύει στο δακτύλιο μέχρι να μπορέσει ο συγκεκριμένος επεξεργαστής να το λάβει. Επίσης κάποιο μήνυμα, υπό κάποιες συνθήκες μπορεί να είναι χρήσιμο σε περισσότερους από ένα επεξεργαστές και με αυτήν την τοπολογία αποφεύγεται η αχρείαστη αποστολή του ίδιου μηνύματος. Για παράδειγμα ο P2 και ο P3 μπορεί να θέλουν να διαβάσουν την ίδια μεταβλητή X, την οποία έχει αποθηκευμένη στην τοπική του μνήμη ο P1. Όταν στείλει το μήνυμα με την μεταβλητή X ο P1, τότε μπορούν να πάρουν τα δεδομένα και ο P2 και ο P3. Αυτό όμως δε θα ήταν δυνατό αν ένας από τους δυο επεξεργαστές ήθελε να γράψει στην μεταβλητή X. Σε αυτές τις περιπτώσεις, ο αλγόριθμος δακτυλίου που υλοποιήθηκε, θα καθορίσει την συμπεριφορά των επεξεργαστών. Κάποια προϊόντα που χρησιμοποιούν τοπολογία δακτυλίου είναι το IBM POWER4 (**P**erformance **O**ptimization **W**ith **E**nhanced **R**ISC) [15], POWER5 [15] και Cell [15]

Η τοπολογία δακτυλίου έχει όμως και κάποιο μειονεκτήματα. Το κυριότερο από αυτά είναι το γεγονός ότι σε περίπτωση βλάβης ενός επεξεργαστή, αυτό θα έχει καταστροφικές συνέπειες για ολόκληρο το δακτύλιο, αφού σπάζει η ροή μηνυμάτων και έτσι ολόκληρο το σύστημα θα καταρρεύσει.

4.2.2 Συνοχή μνήμης σε δακτύλιο

Σε ένα δακτύλιο όταν υπάρχουν πολλαπλοί ταυτόχρονοι αποστολείς μηνυμάτων, τότε μπορεί δυο επεξεργαστές να λάβουν τα μηνύματα με διαφορετική σειρά (Εικόνα 4.2) , και αυτό να προκαλέσει προβλήματα στη συνοχή της μνήμης των τοπικών επεξεργαστών. Για παράδειγμα στην εικόνα ο επεξεργαστής P1 θα λάβει τα μηνύματα με σειρά {A, B} ενώ ο P3 θα τα λάβει με σειρά {B, A}. Το πρόβλημα θα μπορούσε να λυθεί αν επιτρεπόταν σε μόνο ένα επεξεργαστή να στείλει μήνυμα σε κάθε χρονική στιγμή. Αυτό μπορεί να γίνει με την χρησιμοποίηση ενός token το οποίο θα ταξιδεύει στο δακτύλιο, και μόνο ο κάτοχος του να μπορεί να στείλει μήνυμα, και αφού τελειώσει με την αποστολή να προωθεί το token στον επόμενο επεξεργαστή. Με αυτό τον τρόπο μπορούμε να είμαστε σίγουροι ότι όλοι οι επεξεργαστές λαμβάνουν τα μηνύματα με την ίδια σειρά. Αυτή η λύση όμως δεν είναι

καθόλου αποδοτική γιατί το κόστος της αναμονής από κάποιο επεξεργαστή για να στείλει ένα μικρό μήνυμα ελέγχου, επηρεάζει αρνητικά σε μεγάλο βαθμό την απόδοση ενός δακτυλίου. Για αυτό το ενδιαφέρον εστιάζεται σε υλοποιήσεις αλγορίθμων που επιτρέπουν την πολλαπλή και ταυτόχρονη αποστολή μηνυμάτων.



Εικόνα 4.2 *Με την ύπαρξη πολλαπλών αποστολέων, η σειρά παραλαβής των μηνυμάτων εξαρτάται από την θέση του επεξεργαστή*

Κεφάλαιο 5

Αλγόριθμος Greedy Order

5.1 Εισαγωγή

Ο αλγόριθμος Greedy Order [15] είναι ένας από τους αλγόριθμους που επιβάλλουν συνοχή στις μνήμες πολυπύρηνων επεξεργαστών που είναι ενωμένοι σε τοπολογία δακτυλίου. Υιοθετεί τις καταστάσεις, τα δικαιώματα και τις αμετάβλητες συνθήκες του Πίνακα 4.1. Κάθε επεξεργαστής μπορεί να εκδώσει κάποιο αίτημα για εγγραφή (getm) ή ανάγνωση (gets), το οποίο γίνεται αμέσως ενεργό και μπορεί να ικανοποιηθεί από τον επεξεργαστή που θα το λάβει πρώτος και θα έχει στην τοπική του μνήμη τα ζητούμενα δεδομένα. Κάποιοι άλλοι αλγόριθμοι απαιτούν όπως τα αιτήματα περάσουν από κάποιο σημείο ελέγχου προτού γίνουν ενεργά όπως για παράδειγμα ο αλγόριθμος Ordering Point [15]. Το γεγονός αυτό, ότι δηλαδή τα αιτήματα σε αυτό τον αλγόριθμο γίνονται ενεργά μόλις εκδίδονται και ικανοποιούνται από τον πιο κοντινό επεξεργαστή, έχει σαν αποτέλεσμα την μείωση του εύρους ζώνης (bandwidth), αφού δεν χρειάζονται επιπλέον hops για να φτάσουν σε κάποιο σημείο ελέγχου, αλλά και την μείωση της καθυστέρησης στην έκδοση απαντήσεων. Το μεγαλύτερο μειονέκτημα του αλγορίθμου αυτού είναι ότι μπορεί να υπάρχουν συγκρούσεις ανάμεσα στα αιτήματα που εκδίδουν οι επεξεργαστές, και έτσι κάποιος επεξεργαστής μπορεί να αναγκαστεί να επανεκδώσει κάποιο αίτημα πολλές φορές, πράγμα που σε ακραίες περιπτώσεις μπορεί να οδηγήσει σε λιμό (starvation). Για αποφυγή του φαινομένου αυτού, εξετάστηκαν κάποιοι μηχανισμοί, για παράδειγμα οι επανεκδώσεις αιτημάτων να γίνονται με τυχαία σειρά, γεγονός το οποίο αυξάνει την πιθανότητα επιτυχίας, αλλά δεν εγγυείται την αποφυγή του starvation. Άλλη λύση που εξετάστηκε ήταν να προσαρμόσουμε προτεραιότητα στα αιτήματα με βάση το χρόνο (ηλικία) του αιτήματος, αλλά και πάλι αυτό δεν εγγυείται τη

λύση του προβλήματος, γιατί θα απαιτούσε από τον πυρήνα να θυμάται τα αναπάντητα αιτήματα, ή να περιμένει μέχρι να μαζευτούν αρκετά αιτήματα έτσι ώστε να μπορέσει να δώσει κάποια προτεραιότητα στους αιτητές [15].

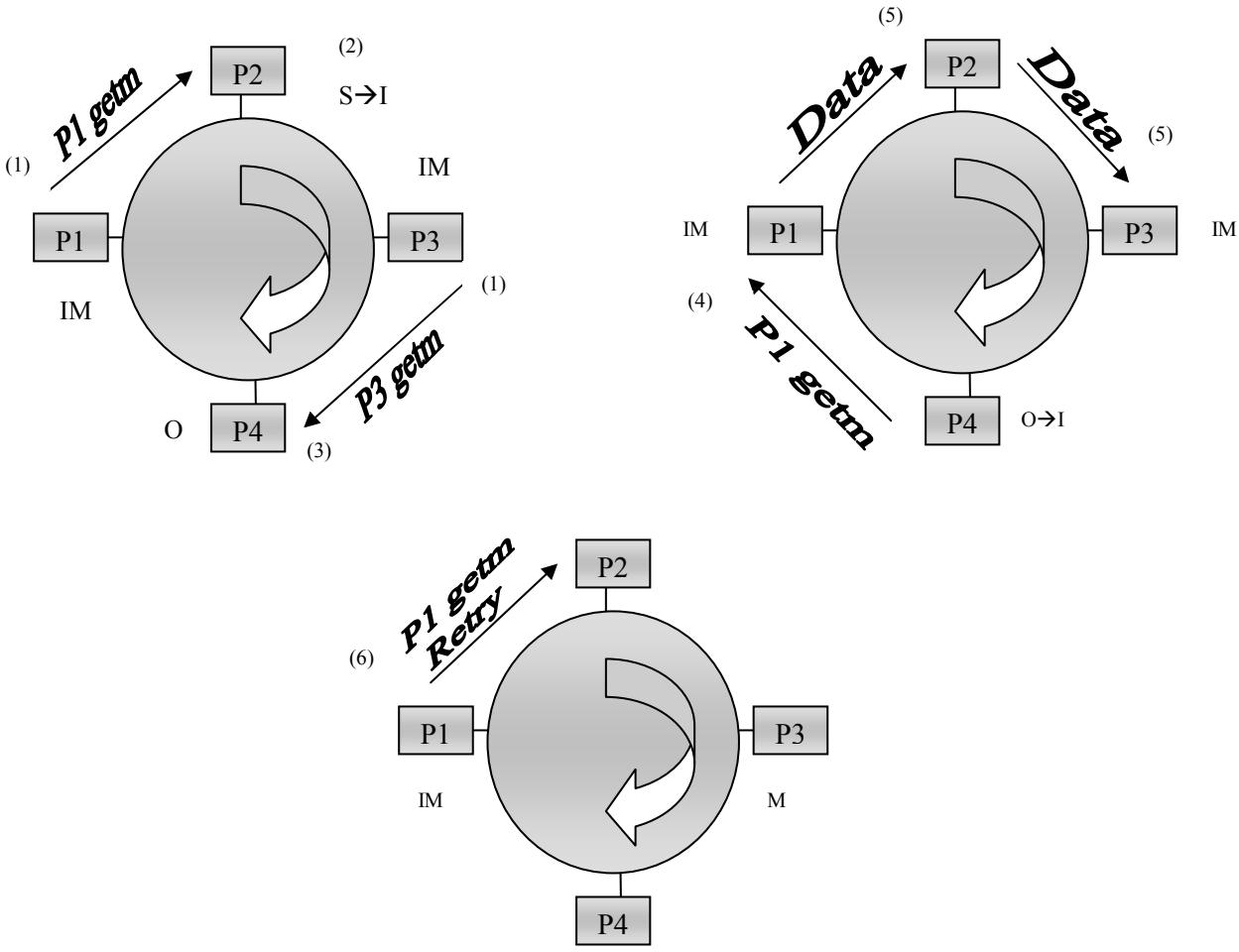
5.2 Περιγραφή Αλγορίθμου

Όπως έχει προαναφερθεί, υπάρχουν δυο ειδών αιτήματα που μπορεί ο κάθε επεξεργαστής να εκδώσει. Είναι τα αιτήματα ανάγνωσης (gets request), και εγγραφής (getm request). Ένα αίτημα ανάγνωσης προσπαθεί να βρει τον ιδιοκτήτη του κομματιού μνήμης που ζητείται, δηλαδή τον επεξεργαστή που το έχει στην τοπική του μνήμη και η κατάσταση του επιτρέπει την ανάγνωση (καταστάσεις M, E, O, S).

Ένα αίτημα εγγραφής εκτελεί αντίστοιχα την ίδια ενέργεια, σε διαφορετικές καταστάσεις{M,E,O} αλλά επιπρόσθετα ακυρώνει όλα τα δικαιώματα που μπορεί να είχαν οι άλλοι επεξεργαστές πάνω σε αντίγραφα του ιδίου κομματιού μνήμης. Με αυτό τον τρόπο επιβάλλεται στον επεξεργαστή που θα χρειαστεί ξανά το συγκεκριμένο κομμάτι μνήμης, να εκδώσει νέο αίτημα ανάγνωσης ή εγγραφής, και να πάρει το πιο φρέσκο αντίγραφο που είναι και το ορθό.

Οι επεξεργαστές που λαμβάνουν κάποιο αίτημα, ψάχνουν στην τοπική τους μνήμη να βρουν το κομμάτι μνήμης που ζητείται και αν το βρουν προσθέτουν ανάλογη πληροφορία στο μήνυμα αιτήματος. Πληροφορία για το αν έχουν το συγκεκριμένο κομμάτι στην τοπική τους μνήμη, αν είναι οι Owners του κομματιού και θα στείλουν στο μέλλον τα δεδομένα, αν έχουν ακυρώσει την κατάσταση του κομματιού μνήμης σε περίπτωση αιτήματος εγγραφής κλπ. Δηλαδή δεν εκδίδεται νέο μήνυμα για την ενημέρωση του αιτητή ότι θα του αποσταλούν τα δεδομένα, αν και αυτό θα ήταν μια πιθανή λύση. Ο λόγος που ο αλγόριθμος δεν εφαρμόζει αυτή την λύση είναι γιατί θα αυξανόταν αχρείαστα το κόστος απόκρισης και η κίνηση μηνυμάτων μέσα στο δακτύλιο. Σύμφωνα με το αλγόριθμο η πληροφορία αυτή προστίθεται στο μήνυμα αιτήματος, δημιουργώντας μια συνδυασμένη απάντηση (combined response) προς τον αιτητή. Η απάντηση αυτή όμως δεν περιέχει τα δεδομένα που ζητήθηκαν. Δείχνει απλά ότι το αίτημα παραλήφθηκε και έτυχε επεξεργασίας. Ο κάθε επεξεργαστής, αφού παραλάβει το αίτημα, το αναλύσει και κάνει τις απαραίτητες ενέργειες, προωθεί το αίτημα στον επόμενο επεξεργαστή, μέχρι το αίτημα να φτάσει πίσω στον επεξεργαστή που το έκδωσε.

Ένα παράδειγμα του τρόπου λειτουργίας του αλγορίθμου, απεικονίζεται στην Εικόνα 5.1. Το παράδειγμα αυτό περιγράφει μια περίπτωση σύγκρουσης αιτημάτων εγγραφής που έχουν αναφερθεί στην εισαγωγή, με αποτέλεσμα την επανέκδοση του ίδιου αιτήματος από τον ίδιο επεξεργαστή. Δυο επεξεργαστές P1 και P3 εκδίδουν ταυτόχρονα ένα αίτημα εγγραφής για το ίδιο κομμάτι μνήμης, το οποίο κατέχει ο P4 σε κατάσταση O. Τα δυο αιτήματα ταξιδεύουν ταυτόχρονα στο δακτύλιο. Ο P2 παραλαμβάνει το αίτημα του P1 και ελέγχει στην τοπική του μνήμη για να δει αν υπάρχει το συγκεκριμένο block, και αφού το βρίσκει τότε ακυρώνει το δικό του αντίγραφο (S→I). Ο P4 παραλαμβάνει το αίτημα του P3 και προσθέτει τις κατάλληλες πληροφορίες στο μήνυμα αιτήματος(δημιουργείται combined response) έτσι ώστε με την επιστροφή του στον P3, να ξέρει ότι θα του αποσταλούν τα δεδομένα. Ο P4 αφού στείλει την συνδυασμένη απάντηση στο αίτημα εγγραφής, αμέσως μετά θα στείλει το μήνυμα απάντησης στον P3 με τα δεδομένα που του ζητήθηκαν και την ίδια στιγμή θα ακυρώσει το δικό του αντίγραφο (O→I). Ο P3 λαμβάνει το αίτημα του P1, ψάχνει στην τοπική του μνήμη για να δει αν κατέχει κάποιο αντίγραφο του κομματιού μνήμης άλλα δεν το βρίσκει (έχει δημιουργήσει αίτημα πριν για το ίδιο κομμάτι μνήμης), και έτσι προωθεί το αίτημα στον επόμενο επεξεργαστή. Ο P4 έχει ήδη ακυρώσει το αντίγραφό του και προωθεί το αίτημα στον επόμενο επεξεργαστή. Ο P1 που έχει ήδη λάβει πριν την συνδυασμένη απάντηση από τον P4 προς τον P3 και έχει καταλάβει ότι εκκρεμεί αίτημα για το κομμάτι μνήμης που ζητά, λαμβάνει πίσω το δικό του αίτημα εγγραφής και αναγκάζεται να επανεκδώσει το ίδιο αίτημα.



***Επεξηγήσεις**:* getm = get modified, M = Modified, O = Owned, S = Shared, I = Invalid, IM = issued request for Modify

(1) Οι επεξεργαστές P1 και P3 εκδίδουν αίτημα εγγραφής (getm) σε block που κατέχει ο P4
(2) Ο P2 παραλαμβάνει το αίτημα και αλλάζει την κατάσταση του αντιγράφου που κατέχει από S(shared) σε I (invalid)
(3) Ο P4 παραλαμβάνει το αίτημα του P3 και δεσμεύεται να του στείλει τα δεδομένα προσθέτοντας δεδομένα στο μήνυμα αιτήματος έτσι ώστε να καταλάβει ο P3 ότι θα του αποσταλούν τα δεδομένα (combined response)
(4) Το αίτημα του P1 προωθείται από τους επεξεργαστές P3 και P4
(5) Ο P3 παραλαμβάνει τα δεδομένα από τον P4 και ολοκληρώνει το αίτημα του
(6) Ο P1 απομακρύνει το αίτημα του από το δακτύλιο και το επανεκδίδει επειδή δεν έλαβε μήνυμα ότι θα του αποσταλούν τα δεδομένα

Εικόνα 5.1 *Παράδειγμα Αλγορίθμου Greedy Order*

Πιο λεπτομερή περιγραφή της αλλαγής των καταστάσεων ανάλογα με την ενέργεια που εκτελείται περιγράφεται στον Πίνακα 5.1 που ακολουθεί. Η πρώτη στήλη του πίνακα αντιστοιχεί στις καταστάσεις που μπορεί να εισέλθει κάποιο κομμάτι μνήμης {I, S, E, O, M}, και η πρώτη γραμμή αντιστοιχεί στα γεγονότα που μπορεί να γίνουν η αιτία για να εκτελεστούν ενέργειες από τον επεξεργαστή και να αλλάξουν οι καταστάσεις των κομματιών μνήμης του. Οι υπόλοιπες γραμμές είναι οι ενέργειες που εκτελεί ο κάθε επεξεργαστής και οι καταστάσεις που θα εισέλθει το ανάλογο κομμάτι μνήμης.(για παράδειγμα το / I δείχνει ότι η νέα κατάσταση του block θα είναι Invalid). Στην στήλη Load όταν η κατάσταση του block είναι I τότε ο επεξεργαστής θα στείλει στο δακτύλιο αίτημα για ανάγνωση (Gets) και θα αλλάξει την κατάσταση του σε Shared. Στις υπόλοιπες καταστάσεις {S, E, O, M}, του επιτρέπεται να διαβάσει το δικό του αντίγραφο. Στην στήλη Store όταν η κατάσταση είναι μια εκ των {I,S,O}, τότε ο επεξεργαστής πρέπει να στείλει αίτημα εγγραφής και να αλλάξει την κατάσταση του block σε Modified, ενώ όταν η κατάσταση είναι {E ή M}, μπορεί να γράψει στο αντίγραφο που βρίσκεται στην τοπική του μνήμη. Στην στήλη Other Getm, όταν ο επεξεργαστής λάβει κάποιο αίτημα εγγραφής από κάποιον άλλο επεξεργαστή, τότε αν η κατάσταση του αντιγράφου του είναι μια εκ των {E, O, M} τότε προωθεί το combined response και ακολούθως δημιουργεί μήνυμα απάντησης με τα ζητούμενα δεδομένα και αλλάζει την κατάσταση του αντιγράφου του σε Invalid. Αν η κατάσταση του αντιγράφου του είναι {I ή S}, τότε απλά αλλάζει την κατάσταση του αντιγράφου του σε Invalid. Τέλος, στη στήλη Other Gets, όταν ο επεξεργαστής λάβει κάποιο αίτημα ανάγνωσης και η κατάσταση του αντιγράφου είναι μια εκ των {E, M, O} τότε προωθεί το combined response και ακολούθως δημιουργεί μήνυμα απάντησης με τα ζητούμενα δεδομένα και αλλάζει την κατάσταση του σε Owned. Αν η κατάσταση του αντιγράφου είναι {S}, τότε απλά προωθεί το combined response και αν είναι {I} τότε δεν γίνεται καμιά ενέργεια.

	Load	**Store**	**Other Getm**	**Other Gets**
I	Send Gets / S	Send Getm / M	/ I	x
S	Do load	Send Getm / M	/ I	Ack Gets
E	Do load	Do store / M	Ack Getm, send data / I	Ack Gets, send data / O
O	Do load	Send Getm / M	Ack Getm, send data / I	Ack Gets, send data
M	Do load	Do store / M	Ack Getm, send data / I	Ack Gets, send data / O

Πίνακας 5.1 *Αλλαγή καταστάσεων ανάλογα με την ενέργεια (x = don't care)*

5.3 Υλοποίηση Αλγορίθμου

Μετά την μελέτη και κατανόηση του αλγορίθμου, το επόμενο βήμα ήταν η μοντελοποίηση του στην Promela. Αρχικά έπρεπε να γίνει μια περιγραφή για τις διεργασίες, τα κανάλια και τις μεταβλητές των μηνυμάτων που θα μεταφέρουν για την επικοινωνία μεταξύ των διεργασιών, και για το πώς θα αναπαρασταθούν οι τοπικές μνήμες και τα κομμάτια μνήμης (blocks).

Κάθε επεξεργαστής με την τοπική του μνήμη θα μοντελοποιείται μέσω μιας διεργασίας, η οποία θα μπορεί να δημιουργεί και να προωθεί αιτήματα, να λαμβάνει αιτήματα από άλλους επεξεργαστές, να τα επεξεργάζεται να τα προωθεί και να εκδίδει απαντήσεις όταν χρειάζεται, και τέλος να λαμβάνει και απαντήσεις σε αιτήματα του ίδιου του επεξεργαστή ή άλλων επεξεργαστών. Η δομή του κάθε επεξεργαστή είναι η ακόλουθη:

```
        active proctype Process1(){
        do
(1)     ::Send!request
(2)     ::Receive?request
(3)                 Process Request
(4)                 Forward request
(5)                 Issue response //when block asked is found valid in cache
(6)     ::Receive?Response
(7)                 Store data      //Response for this processor
                    OR
(8)                 forward respose //Response for other Processor
        }
```

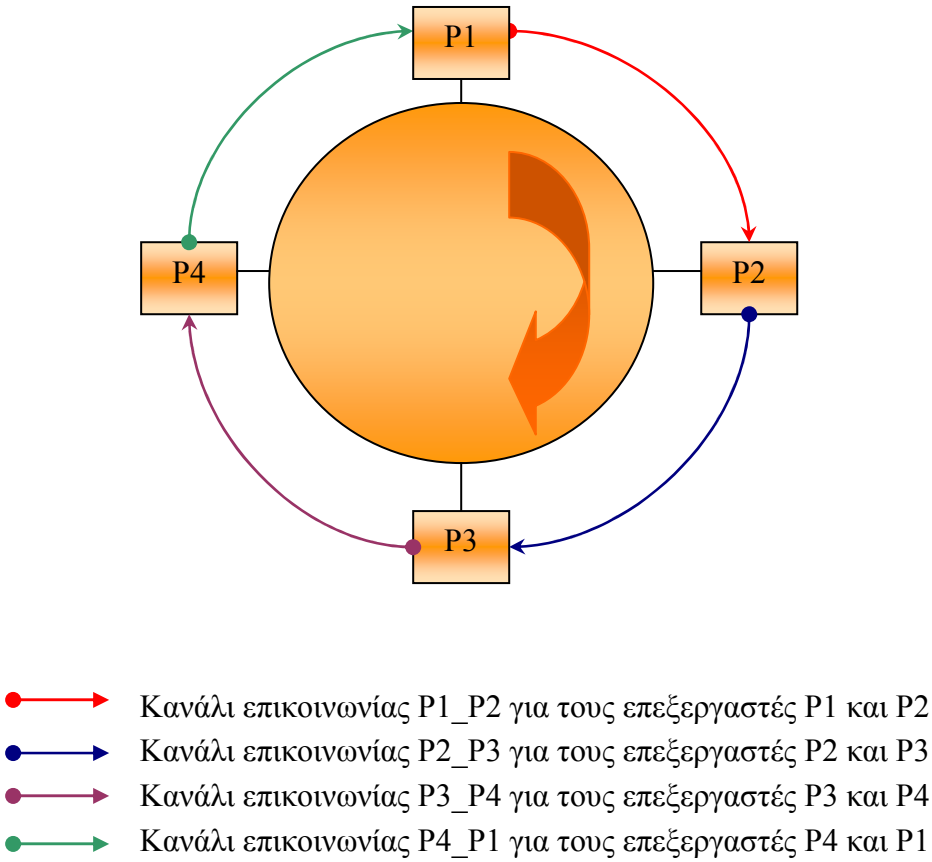
Κατά την υλοποίηση, αντιμετωπίστηκαν δυσκολίες στον συγχρονισμό των καναλιών για την επικοινωνία των επεξεργαστών, και αντιμετωπίζαμε συνεχώς αδιέξοδο (deadlock). Αυτό συνέβαινε γιατί όταν κάποιος επεξεργαστής προωθούσε κάποιο μήνυμα αιτήματος (Γραμμή 4), και στην συνέχεια δημιουργούσε μήνυμα απάντησης (Γραμμή 5), ο επόμενος επεξεργαστής ο οποίος λάμβανε το μήνυμα αιτήματος και το επεξεργαζόταν (Γραμμές 2 και 3), δεν ήταν σε θέση να λάβει το μήνυμα απάντησης (Γραμμή 6) γιατί ασχολείτο ακόμα με το προηγούμενο μήνυμα. Για την αντιμετώπιση αυτού του προβλήματος

χρησιμοποιήσαμε την εντολή "timeout". Όπως έχει ήδη εξηγηθεί, η εντολή αυτή επιτρέπει στο σύστημα να ξεφύγει από τυχόν αδιέξοδο (deadlock). Σε περίπτωση που καμία εντολή του μοντέλου δεν μπορεί να εκτελεστεί, τότε ενεργοποιείται και εκτελείται η εντολή "timeout" η οποία παίρνει τιμή 1 όταν καμία άλλη εντολή στο σύστημα δεν μπορεί να εκτελεστεί και 0 σε όλες τις άλλες περιπτώσεις. Κατά την επαλήθευση του μοντέλου για πιθανά αδιέξοδα, δεν εντοπίστηκαν λάθη και έτσι προχωρήσαμε στην επαλήθευση ιδιοτήτων χρονικής λογικής. Η πρώτη ιδιότητα που ελέγξαμε ήταν για την επαλήθευση ότι δεν χάνονται μηνύματα στον δακτύλιο. Το SPIN όμως εντόπισε λάθος, και έδωσε κάποιο αντιπαράδειγμα. Στο αντιπαράδειγμα αυτό παρατηρήσαμε ότι όταν το μοντέλο έφτανε σε αδιέξοδο, η εντολή "timeout" άφηνε μεν το σύστημα να προχωρήσει, αλλά το μήνυμα του οποίου η αποστολή δημιουργούσε το αδιέξοδο, δεν προωθείτο, αλλά αντιθέτως χανόταν. Το πρόβλημα εντοπίστηκε στο γεγονός ότι όταν προωθείτο το αίτημα στον επόμενο επεξεργαστή (4) και ακολούθως δημιουργείτο απάντηση στο αίτημα(5), ο επόμενος επεξεργαστής μπορεί να μην ήταν σε θέση να λάβει την απάντηση(6), γιατί ήταν στο σημείο που επεξεργαζόταν το αίτημα που προηγουμένως είχε λάβει(2-3). Και επειδή τα κανάλια που χρησιμοποιούσαμε ήταν συγχρονισμένα, τότε τα μηνύματα έπρεπε να ανταλλάζονται απευθείας. Για να το αντιμετωπίσουμε αυτό, αποφασίσαμε να τροποποιήσουμε κάπως τον αλγόριθμο, και αντί για δυο ήδη μηνυμάτων, χρησιμοποιήθηκε ένα. Με αυτό τον τρόπο κάθε επεξεργαστής ήταν σε θέση και να λάβει και να στείλει μήνυμα ταυτόχρονα, λύνοντας το πρόβλημα του αδιεξόδου. Δηλαδή η δομή του επεξεργαστή τροποποιήθηκε ως εξής:

```
active proctype Process1(){
do
::Send!request
::Receive?message
        If message==request then
                Process request
        Else
                Process response
}
```

Βασιστήκαμε στον τύπο μηνύματος για αιτήματα, και προσθέσαμε ακόμα δυο πεδία. Ένα δυαδικό ψηφίο (bit) από το οποίο διακρίναμε αν το μήνυμα ήταν αίτημα ή απάντηση σε αίτημα(0 ή 1 αντίστοιχα), και ακόμα έναν ακέραιο (integer) στο οποίο θα μεταφέραμε τα ζητούμενα δεδομένα. Πιο ακριβής περιγραφή των μηνυμάτων και τον τρόπο που υλοποιήθηκε ο αλγόριθμος θα δοθεί στην συνέχεια.

Η αρχιτεκτονική του δακτυλίου είναι η ακόλουθη:



Εικόνα 5.2: *Αρχιτεκτονική αλγορίθμου*

Η Εικόνα 5.2 δείχνει την αρχιτεκτονική που υλοποιήσαμε. Κάθε επεξεργαστής δέχεται μηνύματα (αιτήματα και απαντήσεις) από τον προηγούμενο του και στέλνει στον επόμενο του μέσω ενός καναλιού.

5.3.1 Υλοποίησης Καναλιών

Όπως είπαμε, χρησιμοποιήσαμε ένα είδος καναλιού, με το οποίο θα επικοινωνούν οι επεξεργαστές:

chan P1_P2_mess_chan = [0] of {bit,int,byte,int,int,bit,int}
//bit = τύπος μηνύματος (0 για ανάγνωση, 1 για εγγραφή)
//int = ταυτότητα αιτητή
//byte = μεταβλητή αιτήματος – χρησιμοποιούμε τα γράμματα του αγγλικού αλφάβητου σε ascii μορφή με πεδίο τιμών 65..90
//int = ταυτότητα κατόχου (owner id)
//int = μεταβλητή που δηλώνει αν θα σταλούν τα δεδομένα στον αιτητή
//bit = αναγνωριστικό μηνύματος (0 για αίτημα, 1 για απάντηση)
//int = τιμή της μεταβλητής

Κάθε επεξεργαστής χρησιμοποιεί ένα δικό του τέτοιο κανάλι για να στείλει μήνυμα στον επόμενο του, και χρησιμοποιεί το κανάλι του προηγούμενου του για να λάβει μηνύματα από αυτόν. Για παράδειγμα σε ένα μοντέλο με τρεις επεξεργαστές ο P1 θα χρησιμοποιεί το κανάλι P1_P2 για να στείλει στον P2 και το P3_P1 για να λάβει από τον P3. Άρα ο αριθμός των καναλιών που θα δηλώσουμε θα είναι ίσος με τον αριθμό των επεξεργαστών του μοντέλου μας. Όπως φαίνεται και από την δήλωση του καναλιού πιο πάνω, χρησιμοποιούμε συγχρονισμένο κανάλι ([0]), και αυτό σημαίνει πως όταν κάποιος επεξεργαστής στείλει κάποιο μήνυμα στον επόμενο του, τότε αυτός θα πρέπει να είναι σε θέση να το λάβει, και αυτό συμβαίνει γιατί κάθε επεξεργαστής είναι σε θέση και να στείλει και να λάβει ταυτόχρονα. Αυτό το γεγονός έλυσε και το πρόβλημα του αδιεξόδου που αντιμετωπίζαμε.

5.3.2 Υλοποίηση Τοπικής Μνήμης

Για την υλοποίηση της τοπικής μνήμης δημιουργήθηκε ένας σύνθετος τύπος δεδομένων:

```
typedef Cache{
      mtype state;
      byte block;
      int data;

}
```

Το πρώτο πεδίο του πιο πάνω τύπου αφορά την κατάσταση του κάθε κομματιού μνήμης το οποίο όπως εξηγήσαμε σε προηγούμενο κεφάλαιο, μπορεί να είναι ένα από τα {E,S,M,O,I,NP}. Ο τύπος αυτής της μεταβλητής είναι mtype, δηλαδή ένα σύνολο από τιμές που καθορίζονται από τον προγραμματιστή:

```
mtype  = {O,S,M,E,I,NP};      //Possible States of a block
```

Το δεύτερο πεδίο είναι για το όνομα της μεταβλητής ή κομματιού μνήμης (block) για το οποίο αναφέρεται η κατάσταση και τα δεδομένα. Τέλος, το τρίτο πεδίο αφορά τα δεδομένα (τιμή) της μεταβλητής. Για την αναπαράσταση της τοπικής μνήμης δηλώθηκε για κάθε επεξεργαστή ένας πίνακας μεγέθους N του σύνθετου τύπου που περιγράψαμε:

```
#define N 5                    //Cache size
Cache P1 [N];
```

5.3.3 Υλοποίηση Διεργασιών

Χρησιμοποιήθηκε ένα είδος διεργασίας το οποίο μοντελοποιούσε την συμπεριφορά του κάθε επεξεργαστή. Πρώτη δουλειά της διεργασίας αυτής είναι να αρχικοποιήσει την τοπική της μνήμη (cache) με κάποιες μεταβλητές και τις τιμές τους, και ακολούθως να περιμένει μέχρι όλοι οι επεξεργαστές να τελειώσουν αυτή την εργασία. Στη συνέχεια μπορεί να ξεκινήσει η αποστολή, παραλαβή, επεξεργασία και προώθηση μηνυμάτων. Κάθε επεξεργαστής μπορεί ταυτόχρονα και να στείλει και να λάβει κάποιο μήνυμα. Η δομή δηλαδή του κάθε επεξεργαστή είναι η ακόλουθη:

```
active proctype Processor(){
Initialize cache
do
::P1_P2_mess_chan!messtype_out,Pid,block,0,0,0,0;
        //sent request
::P3_P1_mess_chan?messtype_in,requester,dblock,ower_id,send_data,response,data
        //receive message
od;
}
```

Στην πρώτη περίπτωση όπου ο επεξεργαστής στέλνει αίτημα, συμπεριλαμβάνει σε αυτό τον τύπο του αιτήματος, αν δηλαδή πρόκειται για ανάγνωση ή εγγραφή (0 ή 1), την ταυτότητα του ως αιτητής, και το όνομα της μεταβλητής που ζητά. Τα άλλα πεδία (ower_id,send_data,response,data) στέλνονται με αρχική τιμή 0, γιατί δεν τα γνωρίζει ο αιτητής, και βάση αυτών των πεδίων θα πάρει τα δεδομένα που ζητά. Όπως έχουμε προαναφέρει κάθε επεξεργαστής δικαιούται να έχει μόνο ένα αίτημα στον δακτύλιο ανά πάσα στιγμή. Αυτό ελέγχεται με μια μεταβλητή (send) τύπου bit η οποία παίρνει τιμή 0 μόλις ο επεξεργαστής εκδώσει το αίτημα του, και τιμή 1 όταν το λάβει πίσω. Πριν την δημιουργία αιτήματος, ελέγχεται αυτή η μεταβλητή:

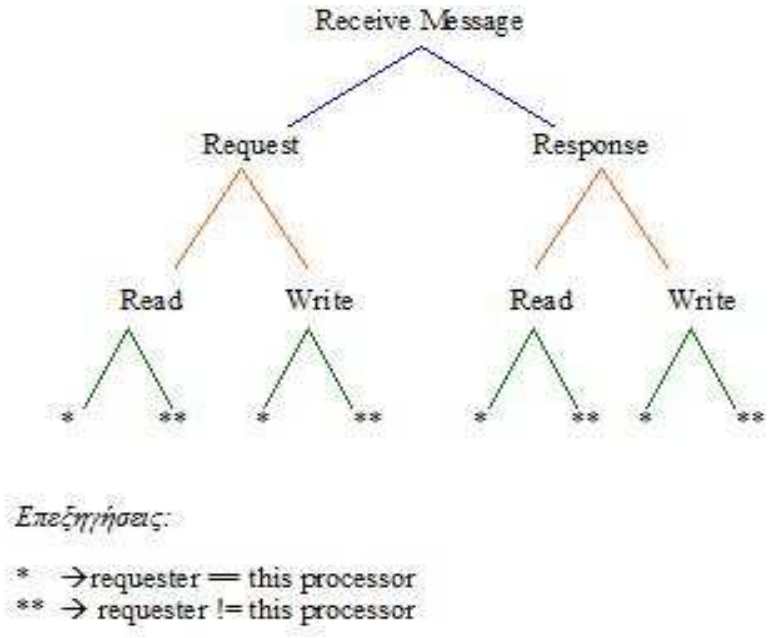
```
do
::atomic{(send==1)->P1_P2Req_mess_chan!messtype_out,P1id,block,0,0,0,0;
          send=0;
         }
                                        .
                                        .
                                        .
```

Αφού σταλεί το αίτημα, ο επεξεργαστής προετοιμάζεται για το επόμενο του αίτημα, αλλάζοντας τον τύπο του αιτήματος και την μεταβλητή την οποία θα ζητήσει. Αυτό το αίτημα θα σταλεί μόνο σε περίπτωση που δεν θα χρειαστεί επανέκδοση του προηγούμενου αιτήματος. Η μεταβλητή MaxVar δηλώνει το μεγαλύτερο όνομα μεταβλητής (γράμμα) που επιτρέπουμε να υπάρχει στο μοντέλο και δηλώνεται σαν καθολική (global) μεταβλητή (για παράδειγμα int MaxVar = 68):

```
.
.
.
block_requested=block;
req_op= messtype_out;
if
      ::block==MaxVar->block=65;
      ::else->block++;
fi;
if
      ::messtype_out==0->messtype_out=1;
      ::else->messtype_out=0;
fi;
```

Στην δεύτερη περίπτωση που ο επεξεργαστής λαμβάνει κάποιο μήνυμα, πρέπει να το αναλύσει και να δει τι είδος μηνύματος είναι (αίτημα ή απάντηση σε αίτημα), ακολούθως να δει αν είναι μήνυμα για ανάγνωση ή εγγραφή, και τέλος να δει αν το μήνυμα απευθύνεται

στον ίδιο ή σε κάποιο άλλο επεξεργαστή και μετά να κάνει τις απαραίτητες ενέργειες (Εικόνα 5.3)



Εικόνα 5.3 *Ανάλυση παραλαβής μηνύματος*

Στις περιπτώσεις που το μήνυμα δεν αφορά τον ίδιο επεξεργαστή (requester!= Processor_id), τότε ο επεξεργαστής εκτελεί τις απαραίτητες ενέργειες και προωθεί το μήνυμα. Μια ενδιαφέρουσα περίπτωση είναι αυτή στην οποία ο επεξεργαστής λαμβάνει μήνυμα για απάντηση μηνύματος εγγραφής το οποίο δεν αφορά τον εαυτό του. Σε αυτή την περίπτωση ελέγχει αν το κομμάτι μνήμης που ζητείται από άλλο επεξεργαστή είναι το ίδιο με αυτό που ζητεί αυτός στο αίτημα του που ταξιδεύει την συγκεκριμένη στιγμή στον δακτύλιο. Αν συμβαίνει αυτό, τότε κάνει την μεταβλητή που ελέγχει πότε πρέπει να γίνει επανέκδοση (re-issue) ίση με 1:

```
if
::block_requested==dblock->
      re_issue=1;
::else->skip;
fi;
```

Ακολούθως ελέγχει την τοπική του μνήμη και ακυρώνει το δικό του αντίγραφο (αν υπάρχει) και προωθεί το μήνυμα. Όταν παραλάβει το δικό του μήνυμα, πρώτα θα ελέγξει την μεταβλητή re-issue και αν είναι ίση με 1 τότε αγνοεί το μήνυμα και το επόμενο του αίτημα

θα αφορά το ίδιο κομμάτι μνήμης. Αν δεν είναι ίση με 1 τότε προχωρεί και αποθηκεύει το κομμάτι μνήμης που του στάλθηκε (αν του στάλθηκε!) και αναλόγως διαβάζει ή γράφει σε αυτό:

```
if
::(requester==P1id)->if
                    ::(re_issue==1)->
                              block=dblock;
                              messtype_out=req_op;
                              re_issue=0;
                    ::else->//1. cache message – read or write
                            //2. Retrieve from memory
```

Αν του στάλθηκε το ζητούμενο κομμάτι μνήμης, τότε το μήνυμα θα έρθει σε μορφή απάντησης (`//1. cache message – read or write`). Αν δεν το έλαβε, τότε το μήνυμα θα επιστρέψει πίσω σαν αίτημα (όπως είχε εκδοθεί) και ο επεξεργαστής θα αναγκαστεί να πάρει το κομμάτι μνήμης που θέλει από την κύρια μνήμη (`//2. Retrieve from memory`).

Ακολουθεί ο κώδικας όπου αποθηκεύεται το κομμάτι μνήμης που λήφθηκε, στην τοπική μνήμη του επεξεργαστή, και οι απαραίτητες αλλαγές όσον αφορά την κατάσταση του κομματιού (ο κώδικας αφορά παραλαβή απάντησης σε αίτημα για ανάγνωση) .

```
if
:: position < N->  P1[position].state = O; //STORE IN NEXT POSITION
                   P1[position].data = data;
                   P1[position].block = dblock;
                   position++;
:: else->position=1;
                   //START STORING FROM FIRST POSITION
                   P1[position].state = O;
                   P1[position].data = data;
                   P1[position].block = dblock;
                   position++;
fi;
```

Πιο κάτω περιγράφεται με ψευδοκώδικα η λειτουργία του αλγορίθμου χρησιμοποιώντας ένα είδος μηνύματος:

Ψευδοκώδικας αλγορίθμου για κάθε επεξεργαστή:

```
//ΔΗΛΩΣΕΙΣ ΚΑΘΟΛΙΚΩΝ ΜΕΤΑΒΛΗΤΩΝ
chan P1_P2Req_mess_chan = [0] of {bit,int,byte,int,int,bit,int}
      //{messtype,requester,data,owner_id,send_data,response,response_data}
```

```
#define N 2                    //Cache size
mtype  = {O,S,M,E,I,NP};       //Possible States of a block
typedef P{
      mtype state;
      byte block;
      int data;
}

P P1[N];
int P1id=1;
```

//ΔΙΕΡΓΑΣΙΑ ΕΠΕΞΕΡΓΑΣΤΗ
active proctype Processor1(){

bit send=1;
bit re_issue=0;

Initialize cache

do

(1):: (send==1)->P1_P2Req_mess_chan!messtype_out,P1id,block,0,0,0,0;
(2) send=0;

//Sent Request for a block

//change block for next request

(3) if messtype_out ==0 then

messtype_out = 1

(4) else

messtype_out = 0 //change type of operation - read or write

(5)::P3_P1Req_mess_chan?messType,requester,dblock,owner_id,send_data,response,data->
//Receive message

(6)If (response==request) then //response == 0

(7) If (messtype==read) then //messtype == 0

(8) If (requester==P1id) then //requester == this Processor

(9) If (forced re-issue) then //re_issue == 1

(10) block = dblock

(11) re_issue = 0

(12) //Next_block_request=current_requesting_block

(13) //discard old request and issue a new one for the same block

Else

(14) Retrieve from Memory //Request came to requester with no

```
                                        //response for data from other processor
                Fi;
(15)            Send=1

(16)        Else  //requester!= this processor
(17)            Check in cache for the requesting block.
(18)        If (blockfound) and (block_state_is_valid) then
(19)            P1_P2Req_mess_chan! messType,requester,dblock,P1id,1,1, P1[i].data
                //issue response.
(20)        Else
(21)  P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
            //If not found forward request
            Fi; //(blockfound)
        Fi;// requester == P1id
    fi; //operation == read

(22)  if (messtype==write) then                 //messtype == 1
(23)        if (requester==P1id) then           //requester == this Processor
(24)            if (forced re-issue) then       //re_issue == 1
(25)                block = dblock
(26)                re_issue = 0
(27)                //Next_block_request=current_requesting_block
(28)                //discard old request and issue a new one for the same block
                Else
(29)                Retrieve from memory //Request came to requester with no
                                        //response for data from other processor

                Fi;
(30)            Send=1
(31)        Else ////requester!= this processor
(32)            If (block==dblock)&&(messType==messtype_out) then
(33)            // Current_Requesting_block==block_in_this_message
(34)                re_issue = 1            //Force Re-issue
(35)                invalidate copy
```

```
(36)    P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
                        //Forward Request
                Else
(37)                    Check in cache.
(38)                If (blockfound) and (block_state_is_valid) then
(39)                    invalidate copy
(40)            P1_P2Req_mess_chan! messType,requester,dblock,P1id,1,1, P1[i].data
                            //issue response.
                        Else
(41)                        invalidate copy – if found
(42)    P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
                            // forward request
                        Fi; //(blockfound)
            Fi; //block==dblock
        Fi;//requester == P1id
    Fi;//messtype==write
Fi; //response == request

(43)  If (response==response_to _request) then          //response==1
(44)    If (messtype==read) then                    //messtype==0
(45)        If (requester==P1id) then               //requester == this Processor
(46)            If (forced re-issue) then           //re_issue == 1
(47)                block = dblock
(48)                re_issue = 0
(49)                //Next_block_request=current_requesting_block
(50)                //discard response and issue a new request for the same block
                Else
(51)                Cache the message. Do read
                Fi;
(52)            Send = 1
            Else //requester!=this Processor
(53)    P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
                        //Forward Response
            Fi; //requester == P1id
```

```
        Fi; //Operation ==read

(54)    If (messtype == write) then                 //messtype == 1
(55)            If (requester==P1id) then            //requester == this processor
(56)                    If (forced re-issue) then    //re_issue == 1
(57)                            block = dblock
(58)                            re_issue = 0
(59)                            //Next_block_request=current_requesting_block
(60)                            //discard response and issue a new request for the same block
                        Else
(61)                            Cache the message. Do write
                        Fi; //forced re-issue
(62)                    Send = 1
(63)            Else //requester != this processor
(64)                    If (block==dblock) then
                        // Current_Requesting_block==block_in_this_message
(65)                            re_issue = 1                    //Force Re-issue
(66)                            invalidate copy
(67)    P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
                        // Forward Response
                        Else
(68)                            Check in cache.
(69)                            invalidate_copy(if any)
(70)    P1_P2Req_mess_chan! messType,requester,dblock,owner_id,send_data,response,data
                                // Forward response
                        Fi; //block==dblock
                Fi; //requester == P1id
        Fi; //operation==write
Fi; //Message==response
}
```

Επεξήγηση ψευδοκώδικα:

Να σημειωθεί ότι κάθε επεξεργαστής μπορεί να έχει μόνο ένα αίτημα ανά πάσα στιγμή στο δακτύλιο. Ξεκινώντας ο κάθε επεξεργαστής μπορεί ταυτόχρονα να στείλει και να λάβει μηνύματα (Γραμμές 1 και 5). Σε περίπτωση που ο επεξεργαστής στείλει ένα μήνυμα αιτήματος(Γραμμή 1), τότε ο επόμενος του το παραλαμβάνει αμέσως (Γραμμή 5), και ξεκινά την επεξεργασία του. Ο αποστολέας δεν δικαιούται να στείλει άλλο αίτημα (Γραμμή 2) μέχρι να λάβει το αίτημα ή την απάντηση του πίσω (Γραμμές 15,30,52 και 62). Μπορεί όμως να παραλαμβάνει, να επεξεργάζεται και να προωθεί αιτήματα άλλων επεξεργαστών.

Επεξεργασία μηνύματος που αφορά αίτημα:

Όταν ένας επεξεργαστής λάβει κάποιο μήνυμα, τότε ελέγχει πρώτα αν πρόκειται για μήνυμα αιτήματος (Γραμμή 6) ή για απάντηση σε αίτημα (Γραμμή 43). Αν πρόκειται για αίτημα, τότε ελέγχει αν είναι αίτημα εγγραφής (Γραμμή 22) ή διαβάσματος (Γραμμή 7). Αν είναι αίτημα για διάβασμα, τότε ελέγχει πρώτα αν ο αιτητής είναι ο εαυτός του (Γραμμή 8). Αν συμβαίνει αυτό, τότε ελέγχει αν πρέπει να κάνει επανέκδοση του αιτήματος του (Γραμμή 9). Αυτό συμβαίνει σε περίπτωση που πριν να λάβει το δικό του αίτημα πίσω, κάποιος άλλος επεξεργαστής έχει ζητήσει το ίδιο κομμάτι μνήμης για εγγραφή (Γραμμές 34 και 65). Αυτό ελέγχεται σε άλλο σημείο του κώδικα. Αν δεν πρέπει να κάνει επανέκδοση, τότε σημαίνει ότι κανένας επεξεργαστής δεν κατέχει το κομμάτι μνήμης που ζητά και θα πρέπει να το ανακτήσει από την κύρια μνήμη (Γραμμή 14). Αν ο αιτητής δεν είναι ο εαυτός του (Γραμμή 16), τότε ελέγχει στην τοπική του μνήμη και ψάχνει να βρει το κομμάτι μνήμης που ζητείται (Γραμμή 17). Αν το βρει και είναι σε ορθή κατάσταση (valid state), τότε προωθεί το μήνυμα σαν απάντηση αιτήματος και όχι σαν αίτηση (αλλάζει το νέο πεδίο ‘response’ που εισαγάγαμε από 0 σε 1) μαζί με τα δεδομένα του κομματιού μνήμης (Γραμμές 18 και 19). Αν δεν το βρει ή το βρει και η κατάσταση του δεν είναι ορθή, τότε απλά προωθεί το αίτημα στον επόμενο επεξεργαστή (Γραμμές 20 και 21).

Αν το αίτημα είναι για εγγραφή (Γραμμή 22), τότε ελέγχει και πάλι αν το αίτημα είναι δικό του (Γραμμή 23). Αν είναι δικό του, τότε ελέγχει αν πρέπει να επανεκδώσει το αίτημα (Γραμμές 24 μέχρι 28). Αν δεν πρέπει να το επανεκδώσει, τότε πρέπει να πάρει τα δεδομένα που θέλει από την μνήμη γιατί το αίτημα του επέστρεψε σαν αναπάντητο αίτημα, και έτσι κανένας επεξεργαστής δεν τα έχει στην τοπική του μνήμη για να του τα στείλει (Γραμμή 29). Αν ο αιτητής δεν είναι ο ίδιος ο επεξεργαστής (Γραμμή 31), τότε ελέγχει πρώτα αν το κομμάτι μνήμης που ζητείται είναι το ίδιο με αυτό που ήδη ζητεί ο επεξεργαστής σε κάποιο

δικό του αίτημα ανάγνωσης που ταξιδεύει την ίδια ώρα στο δακτύλιο (Γραμμή 33). Αν συμβαίνει αυτό, τότε με μια μεταβλητή (re_issue) ειδοποιούμε ότι μόλις ο επεξεργαστής παραλάβει το δικό του μήνυμα πίσω, να το αγνοήσει (Γραμμή 34), και να επανεκδώσει (Force Re-issue) το αίτημα. Ακολούθως ο επεξεργαστής ακυρώνει το αντίγραφο στην τοπική του μνήμη αν υπάρχει και προωθεί το αίτημα στον επόμενο επεξεργαστή (Γραμμή 35). Αν το κομμάτι μνήμης του αιτήματος δεν είναι το ίδιο με το κομμάτι που ζητεί ο επεξεργαστής (Γραμμή 37), τότε ελέγχει στην τοπική του μνήμη, και αν το βρει και είναι σε ορθή κατάσταση, ακυρώνει το δικό του αντίγραφο (Γραμμή 39) και προωθεί το αίτημα σαν απάντηση (Γραμμή 40) (αλλάζει το νέο πεδίο που εισαγάγαμε από 0 σε 1) μαζί με τα δεδομένα,. Αν δεν το βρει τότε απλά προωθεί το αίτημα (Γραμμή 42).

Επεξεργασία μηνύματος που αφορά απάντηση:

Αν το μήνυμα είναι μήνυμα απάντησης (Γραμμή 43), τότε και πάλι ο επεξεργαστής ελέγχει αν είναι απάντηση για αίτημα διαβάσματος (Γραμμή 44) και αν ο αιτητής είναι ο ίδιος ο επεξεργαστής (Γραμμή 45). Αν συμβαίνει αυτό τότε ελέγχει και πάλι αν πρέπει να επανεκδώσει το μήνυμα (Γραμμή 46 μέχρι 50). Αν δεν πρέπει, τότε αποθηκεύει τα δεδομένα στην τοπική του μνήμη και τα διαβάζει (Γραμμή 51). Αν το μήνυμα δεν αφορά τον συγκεκριμένο επεξεργαστή, τότε απλά το προωθεί (Γραμμή 53).

Αν το μήνυμα απάντησης είναι απάντηση σε αίτημα για εγγραφή (Γραμμή 54), τότε ελέγχει αν η απάντηση αφορά τον συγκεκριμένο επεξεργαστή (Γραμμή 55). Αν συμβαίνει αυτό, τότε ελέγχει αν πρέπει να επανεκδώσει το αίτημα του (Γραμμές 56 μέχρι 60). Αν δεν πρέπει, τότε αποθηκεύει τα δεδομένα στην τοπική του μνήμη και γράφει στο συγκεκριμένο κομμάτι μνήμης (Γραμμή 61). Αν το μήνυμα αφορά άλλον επεξεργαστή (Γραμμή 63), τότε ελέγχει πρώτα αν το κομμάτι μνήμης που ζητείται είναι το ίδιο με αυτό που ήδη ζητεί ο επεξεργαστής σε αίτημα που ταξιδεύει την ίδια ώρα στο δακτύλιο (Γραμμή 64). Αν συμβαίνει αυτό, τότε με μια μεταβλητή ειδοποιούμε ότι μόλις ο επεξεργαστής παραλάβει το δικό του μήνυμα πίσω, να το αγνοήσει και να το επανεκδώσει (Γραμμή 65). Στην συνέχεια ο επεξεργαστής ακυρώνει το αντίγραφο στην τοπική του μνήμη αν υπάρχει (Γραμμή 66) και προωθεί την απάντηση στον επόμενο επεξεργαστή (Γραμμή 67). Αν το κομμάτι μνήμης του μηνύματος απάντησης που ζητεί κάποιος άλλος επεξεργαστής δεν είναι το ίδιο με αυτό που ήδη ζητεί ο συγκεκριμένος επεξεργαστής, τότε ελέγχει στην τοπική του μνήμη αν υπάρχει κάποιο αντίγραφο (Γραμμή 68), και αν υπάρχει και είναι ορθό τότε το ακυρώνει (Γραμμή 69)

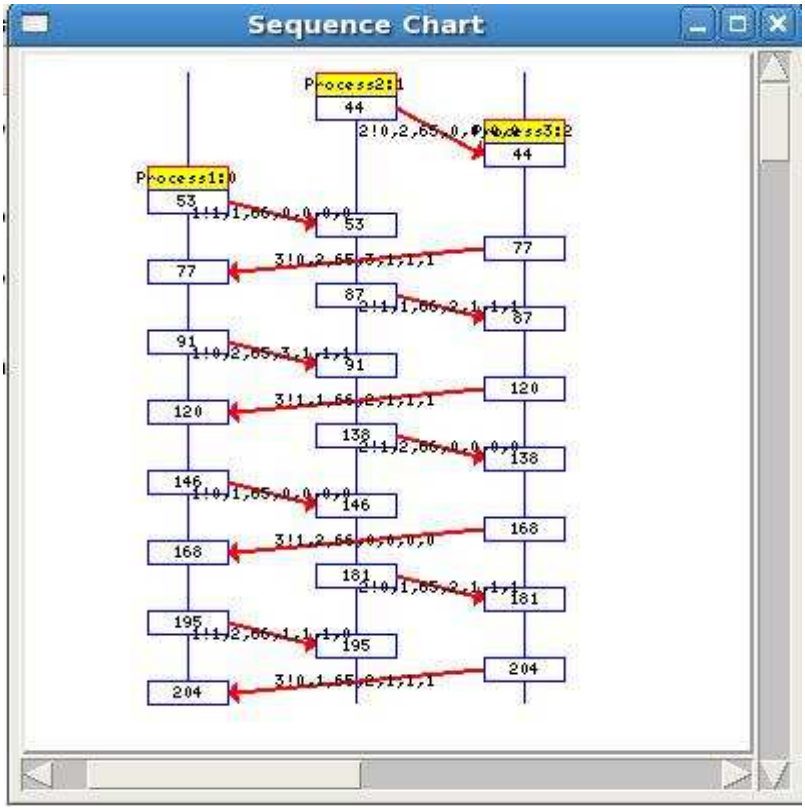
και προωθεί το μήνυμα απάντησης στον επόμενο επεξεργαστή (Γραμμή 70). Αν δεν υπάρχει τότε απλά προωθεί το μήνυμα απάντησης. Ο κώδικας του αλγορίθμου βρίσκεται στο Παράρτημα Α

Κεφάλαιο 6

Αποτελέσματα

6.1 Επαλήθευση Αλγορίθμου για αδιέξοδα

Πριν παρουσιάσουμε τα αποτελέσματα της επαλήθευσης, θα παρουσιάσουμε μια εικόνα (Εικόνα 6.1) η οποία μας δείχνει μέσω μιας προσομοίωσης, και με την βοήθεια του MSC (Message Sequence Chart) τον τρόπο που επικοινωνούν και ανταλλάζουν μηνύματα οι επεξεργαστές σε ένα μοντέλο με τρείς επεξεργαστές.

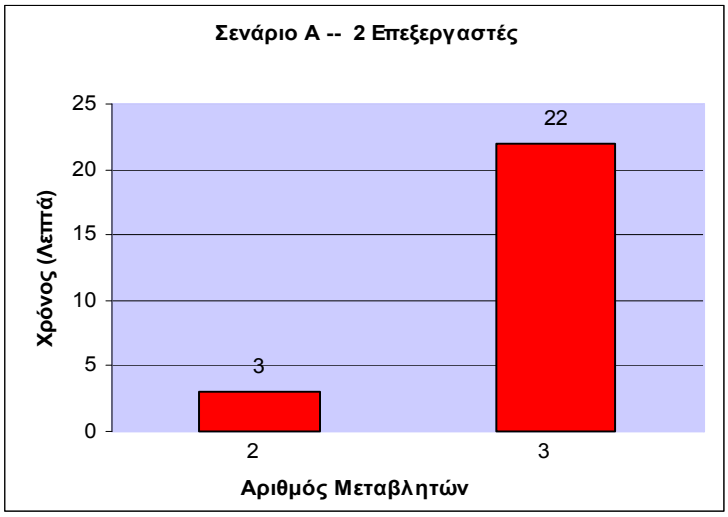


Εικόνα 6.1: *Επικοινωνία επεξεργαστών στο MSC (Message Sequence Chart)*

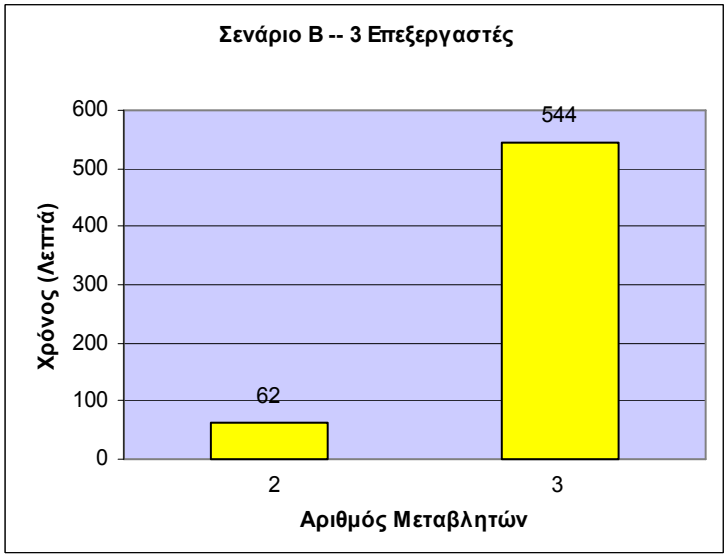
Στην πιο πάνω εικόνα, αρχικά ο επεξεργαστής P2 δημιουργεί ένα αίτημα ανάγνωσης για το κομμάτι 65 (μεταβλητή ΄Α΄) και το προωθεί στον επεξεργαστή P3. Στη συνέχεια ο επεξεργαστής P1 δημιουργεί αίτημα εγγραφής για το κομμάτι 66 (μεταβλητή ΄Β΄) και το προωθεί στον επεξεργαστή P2. Το κομμάτι που ζητά ο P2 βρέθηκε στην μνήμη του P3 και έτσι αυτός προωθεί μήνυμα απάντησης στον με τα δεδομένα στον επεξεργαστή P1. Ο επεξεργαστής P2 βρίσκει το κομμάτι που ζητεί ο P1, ακυρώνει το δικό του αντίγραφο (επειδή πρόκειται για εγγραφή) και προωθεί στον P3 μήνυμα απάντησης με τα δεδομένα. Ο επεξεργαστής P1 προωθεί στον P2 το μήνυμα απάντησης που έλαβε από τον P3 για το αίτημα του P2, και ο P2 αποθηκεύει το κομμάτι 65 (΄Α΄) στην τοπική του μνήμη και διαβάζει τα δεδομένα του κομματιού. Ο επεξεργαστής P3 λαμβάνει τα το μήνυμα απάντησης για εγγραφή από τον P2, ψάχνει στην τοπική του μνήμη για να ακυρώσει το αντίγραφο του συγκεκριμένου κομματιού αν υπάρχει, και ακολούθως προωθεί το μήνυμα απάντησης στον P1, δηλαδή στον αιτητή. Ο P1 παραλαμβάνει το μήνυμα, αποθηκεύει το κομμάτι στην τοπική του μνήμη, και μετά εκτελεί εγγραφή στο συγκεκριμένο κομμάτι. Στην συνέχεια ο P2 δημιουργεί αίτημα εγγραφής για το 66 (μεταβλητή ΄Β΄) και το προωθεί στον P3. Ο P1 δημιουργεί αίτημα για ανάγνωση του 65 (μεταβλητή ΄Α΄) και συνεχίζεται η διαδικασία αποστολής, παραλαβής, επεξεργασίας και προώθησης μηνυμάτων.

Για την επαλήθευση του μοντέλου για πιθανό αδιέξοδο, τρέξαμε δυο διαφορετικά σενάρια. Στο πρώτο σενάριο που τρέξαμε χρησιμοποιήσαμε δυο επεξεργαστές και στο δεύτερο τρεις επεξεργαστές. Διενεργήσαμε εξαντλητική επαλήθευση του μοντέλου με δυο και τρεις μεταβλητές για το κάθε σενάριο, χωρίς να εντοπιστεί οποιοδήποτε αδιέξοδο. Οι χρόνοι επαλήθευσης για τα δυο σενάρια βρίσκονται στους Πίνακες 6.1 και 6.2 αντίστοιχα. Παρατηρούμε την δραματική αύξηση του χρόνου που δημιουργείται με την απλή προσθήκη μιας νέας μεταβλητής, η οποία οφείλεται στην αύξηση του αριθμού καταστάσεων όπως φαίνεται στον Πίνακα 6.3.

Έγινε επίσης προσπάθεια για εξαντλητική επαλήθευση και με τέσσερις επεξεργαστές, αλλά λόγω του ότι δεν αρκούσε η μνήμη, περιοριστήκαμε στην επαλήθευση Bitstate, δηλαδή ελέγχθηκε ένα υποσύνολο του συνόλου καταστάσεων του μοντέλου. Σε αυτήν την επαλήθευση δεν εντοπίστηκε οποιοδήποτε λάθος.



Πίνακας 6.1: *Χρόνοι επαλήθευσης πρώτου σεναρίου με δυο επεξεργαστές*



Πίνακας 6.2: *Χρόνοι επαλήθευσης δεύτερου σεναρίου με τρείς επεξεργαστές*

Σενάριο	Αριθμός Μεταβλητών	Αριθμός Καταστάσεων
A	2	1026732
A	3	6131149
B	2	8086309
B	3	78192317

Πίνακας 6.3: *Αριθμοί Καταστάσεων στα διαφορετικά σενάρια*

6.2 Επαλήθευση ιδιοτήτων χρονικής λογικής

Αφού έγινε με επιτυχία η επαλήθευση του μοντέλου για αδιέξοδα, στην συνέχεια προχωρήσαμε στην επαλήθευση κάποιων ιδιοτήτων χρονικής λογικής μέσω του LTL manager. Οι ιδιότητες ελέγχθηκαν για ένα επεξεργαστή, και αφού ο κώδικας όλων των επεξεργαστών είναι συμμετρικός, συνεπάγεται ότι ισχύει για όλους χωρίς βλάβη της γενικότητας.

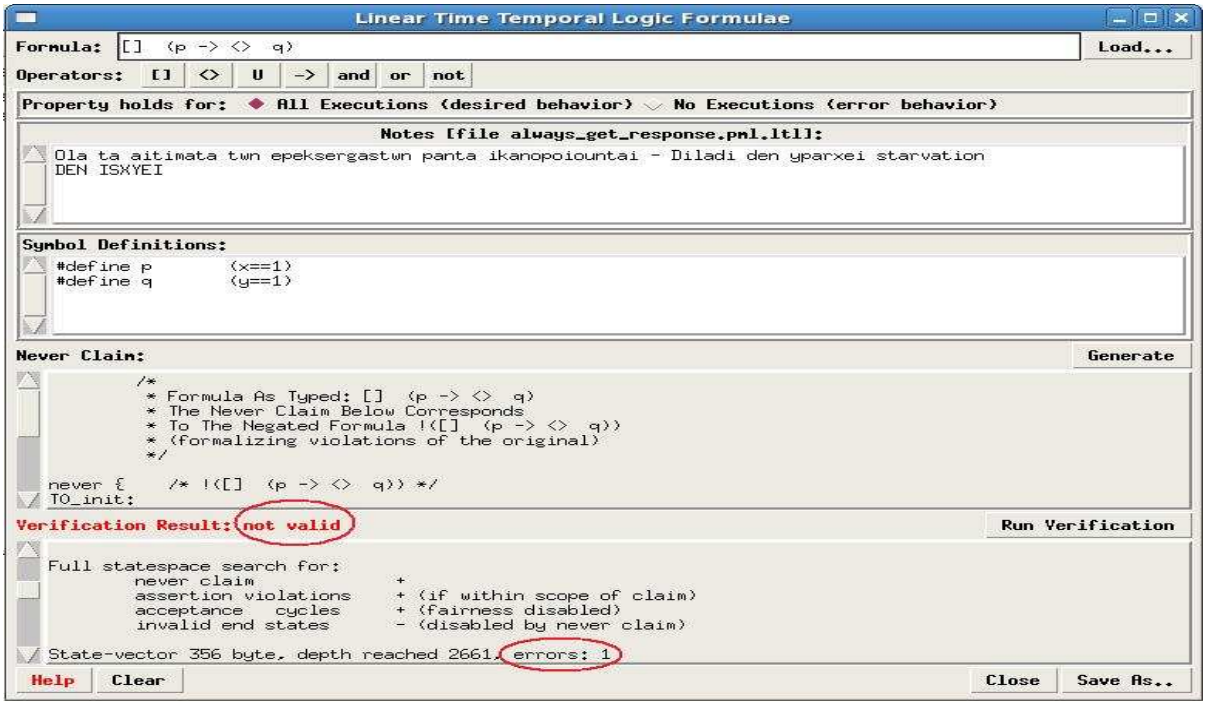
Η πρώτη ιδιότητα που ελέγχθηκε είναι για το αν υπάρχει απώλεια μηνυμάτων στο δακτύλιο. Αν δηλαδή κάποιος επεξεργαστής εκδώσει κάποιο αίτημα, τότε το μήνυμα αυτό θα επιστρέφει πάντα στον επεξεργαστή είτε σαν απάντηση, είτε σαν αίτημα το οποίο δηλώνει ότι είτε πρέπει να γίνει επανέκδοση του αιτήματος, είτε ότι κανένας επεξεργαστής δεν έχει τα ζητούμενα δεδομένα και άρα πρέπει να ληφθούν από την κύρια μνήμη. Η ιδιότητα εκφράστηκε στην LTL ως **[](p-><>q)**. Η πρόταση p εκφράζει το γεγονός ότι ο επεξεργαστής έχει εκδώσει κάποιο αίτημα, και η πρόταση q ότι το αίτημα επέστρεψε στον αιτητή. Η πρόταση p έχει δηλωθεί ως (Process1:send==0) και γίνεται αληθής μόλις η τοπική μεταβλητή send του Process1 γίνει 0 που σημαίνει πως ο επεξεργαστής έχει εκδώσει κάποιο αίτημα. Η πρόταση q έχει δηλωθεί ως (Process1:send==1) και γίνεται αληθής μόλις η τοπική μεταβλητή send του Process1 γίνει 1, που σημαίνει πως ο επεξεργαστής έχει λάβει πίσω το μήνυμα που έστειλε είτε σαν αίτημα, είτε σαν απάντηση σε αίτημα. Ολόκληρη η πρόταση εκφράζει την ιδιότητα ότι πάντα όταν ο επεξεργαστής εκδώσει αίτημα, τότε σε κάποια στιγμή στο μέλλον θα λάβει το μήνυμα του πίσω, που συνεπάγεται πως δεν χάνονται μηνύματα στον δακτύλιο. Η ιδιότητα επαληθεύτηκε χωρίς λάθη σε μοντέλο με δυο επεξεργαστές, και ο χρόνος και ο αριθμός καταστάσεων φαίνεται στον Πίνακα 6.4

Αρ. Επεξεργαστών	Αρ. Μεταβλητών	Χρόνος Επαλήθευσης (Λεπτά)	Αριθμός Καταστάσεων
2	2	12	2209541

Πίνακας 6.4: *Στατιστικά επαλήθευσης πρώτης ιδιότητας*

Η δεύτερη ιδιότητα που ελέγχθηκε αφορούσε το κατά πόσον ένας επεξεργαστής θα παίρνει πάντα απάντηση για το αίτημα του. Η ιδιότητα αυτή εκφράστηκε στην LTL με τον τύπο [](p→<>q). Η πρόταση p εκφράζει το γεγονός ότι έχει εκδοθεί κάποιο αίτημα για κάποιο

κομμάτι μνήμης και ελέγχεται με μια μεταβλητή x η οποία παίρνει τιμή 1 όταν εκδοθεί το αίτημα. Η πρόταση q εκφράζει το γεγονός ότι το αίτημα απαντήθηκε, ή ότι ο επεξεργαστής θα πρέπει να πάρει τα δεδομένα από την μνήμη και ελέγχεται με μια μεταβλητή y η οποία παίρνει τιμή 1 όταν λάβουμε την απάντηση για το αίτημα. Με άλλα λόγια η πρόταση ολόκληρη εκφράζει την ιδιότητα ότι πάντα όταν ένας επεξεργαστής εκδώσει κάποιο αίτημα, τότε κάποτε θα λάβει κάποια απάντηση και δε θα χρειαστεί να επανεκδίδει το αίτημα του άπειρες φορές. Την ιδιότητα αυτή προσπαθήσαμε αρχικά να την επαληθεύσουμε σε μοντέλο με τρείς επεξεργαστές και δυο μεταβλητές. Ο LTL manager εντόπισε λάθος σε αυτή την ιδιότητα (Εικόνα 5.7). Αυτό οφείλεται στο γεγονός ότι τα αιτήματα εγγραφής έχουν προτεραιότητα έναντι στα αιτήματα ανάγνωσης, και όταν ένας επεξεργαστής εκδώσει αίτημα για ανάγνωση κάποιου κομματιού, και την ίδια ώρα πάντα εκδίδεται κάποιο αίτημα εγγραφής για το ίδιο κομμάτι μνήμης από κάποιον άλλο επεξεργαστή, τότε ο επεξεργαστής που κάνει αίτημα για ανάγνωση θα επανεκδίδει το αίτημα του για πάντα.



Εικόνα 6.2: *Αποτέλεσμα επαλήθευσης δεύτερης ιδιότητας*

Αυτό συμβαίνει γιατί ο πρώτος επεξεργαστής εκδίδει αίτημα ανάγνωσης για το A(65) , αλλά πάντα υπάρχει και ένας από τους άλλους επεξεργαστές που εκδίδει αίτημα εγγραφής για το ίδιο κομμάτι. Έτσι ο πρώτος επεξεργαστής αναγκάζεται να επανεκδίδει το αίτημα του για πάντα, λόγω της προτεραιότητας των αιτημάτων εγγραφής έναντι στα αιτήματα ανάγνωσης.

Η τρίτη ιδιότητα που ελέγχθηκε, είναι για το αν την ώρα που κάποιος επεξεργαστής εκτελεί εγγραφή σε κάποιο κομμάτι μνήμης, τότε όλα τα άλλα αντίγραφα στις τοπικές μνήμες των άλλων επεξεργαστών, είτε είναι σε κατάσταση Invalid είτε δεν υπάρχουν. Η ιδιότητα αυτή ελέγχθηκε μέσω της πρότασης []((p && q && r && m)→(s || w)). Ο συνδυασμός των προτάσεων p (Process1:messType==1), q (Process1:requester==P1id), r (Process1:response==1) και m(Process1:re_issue==0) , εκφράζουν το γεγονός ότι ο επεξεργαστής που ελέγχεται, εκτελεί εγγραφή σε κάποιο κομμάτι μνήμης. Οι πρόταση s (Process2:state==I) δηλώνει ότι η κατάσταση του αντιγράφου στον επεξεργαστή δυο είναι Invalid, και η πρόταση w (Process2:found==0) δηλώνει ότι το αντίγραφο δεν υπάρχει στον επεξεργαστή. Η ιδιότητα αυτή ελέγχθηκε στο μοντέλο με δυο επεξεργαστές και δυο μεταβλητές και ο χρόνος και ο αριθμός καταστάσεων φαίνεται στον Πίνακα 6.5.

Αρ. Επεξεργαστών	Αρ. Μεταβλητών	Χρόνος Επαλήθευσης (Λεπτά)	Αριθμός Καταστάσεων
2	2	10	1748620

Πίνακας 6.5: *Στατιστικά επαλήθευσης τρίτης ιδιότητας*

Η τελευταία ιδιότητα που ελέγχθηκε, είναι για το αν η κατάσταση των δεδομένων που αποστέλλονται κατά την απάντηση σε κάποιο αίτημα είναι ορθή. Τον έλεγχο της ιδιότητας αυτής την χωρίσαμε σε έλεγχο για αιτήματα εγγραφής και αιτήματα απάντησης. Η ιδιότητα αυτή ελέγχθηκε μέσω της πρότασης []((p && r)-> <>(q || s || t || n)). Ο συνδυασμός των προτάσεων p (Process1:send==0) και r (Process1:messType==1) εκφράζουν το γεγονός ότι εκδόθηκε μήνυμα για εγγραφή (ή ανάγνωση για r (Process1:messType==0)). Οι προτάσεις q (Process2:state==M), s (Process2:state==O), t (Process2:state==E), εκφράζουν τις ορθές καταστάσεις των κομματιών μνήμης κατά τις οποίες μπορεί να δημιουργηθεί κάποιο μήνυμα απάντησης και η πρόταση n (Process2:found==0) δηλώνει ότι το κομμάτι μνήμης που ζητείται δεν βρέθηκε στην τοπική μνήμη του επεξεργαστή. Στους Πίνακες 6.6 και 6.7 φαίνονται οι χρόνοι και οι αριθμοί καταστάσεων για τον έλεγχο για αιτήματα εγγραφής και αιτήματα απάντησης αντίστοιχα

Αρ. Επεξεργαστών	Αρ. Μεταβλητών	Χρόνος Επαλήθευσης (Λεπτά)	Αριθμός Καταστάσεων
2	2	11	2476249

Πίνακας 6.6: *Στατιστικά επαλήθευσης τέταρτης ιδιότητας (για αιτήματα εγγραφής)*

Αρ. Επεξεργαστών	Αρ. Μεταβλητών	Χρόνος Επαλήθευσης (Λεπτά)	Αριθμός Καταστάσεων
2	2	5	1211198

Πίνακας 6.7: *Στατιστικά επαλήθευσης τέταρτης ιδιότητας (για αιτήματα ανάγνωσης)*

Κεφάλαιο 7

Συμπεράσματα

7.1 Συμπεράσματα

Ο κύριος στόχος αυτής της εργασίας ήταν η μοντελοποίηση, η ανάλυση και η επαλήθευση αλγορίθμου συνοχής μνήμης σε πολυπύρηνους επεξεργαστές, οι οποίοι είναι ενωμένοι σε τοπολογία δακτυλίου, χρησιμοποιώντας το εργαλείο SPIN. Έγινε για πρώτη φορά επαλήθευση του αλγορίθμου Greedy Order [15] για εντοπισμό αδιεξόδων και επίσης επαληθεύτηκαν ορισμένες ιδιότητες χρονικής λογικής. Συγκεκριμένα έγινε εξαντλητική επαλήθευση για εύρεση αδιεξόδων σε δυο σενάρια. Το πρώτο σενάριο υλοποιήθηκε με δυο επεξεργαστές και έγινε επαλήθευση χρησιμοποιώντας δυο και τρείς μεταβλητές, ενώ το δεύτερο σενάριο υλοποιήθηκε με τρεις επεξεργαστές και επαληθεύτηκε και πάλι χρησιμοποιώντας δυο και τρεις μεταβλητές. Και τα δυο σενάρια επαληθεύτηκαν με επιτυχία γεγονός που αποδεικνύει πως ο αλγόριθμος δεν πάσχει από αδιέξοδο.

Στην συνέχεια επαληθεύτηκαν κάποιες ιδιότητες χρονικής λογικής. Η πρώτη ιδιότητα αφορούσε την απώλεια μηνυμάτων στον δακτύλιο, καθώς αυτά προωθούνται από επεξεργαστή σε επεξεργαστή. Η ιδιότητα αυτή επαληθεύτηκε χωρίς λάθη, γεγονός που αποδεικνύει πως δεν χάνονται μηνύματα στον δακτύλιο. Η δεύτερη ιδιότητα που ελέγχθηκε ήταν για το αν όλα τα αιτήματα που εκδίδονται, κάποτε ικανοποιούνται. Σε αυτή την ιδιότητα εντοπίστηκε κύκλος όπου κάποιος επεξεργαστής αναγκάζεται να επανεκδώσει το αίτημα του άπειρες φορές. Αυτό αποδεικνύει πως ο συγκεκριμένος αλγόριθμος πάσχει από το πρόβλημα του λιμού (starvation). Η τρίτη ιδιότητα αφορούσε την επαλήθευση της αμετάβλητης συνθήκης που λέει πως όταν κάποιος επεξεργαστής γράφει σε κάποιο κομμάτι

της τοπικής του μνήμης, τότε όλα τα αντίγραφα στις τοπικές μνήμες των άλλων επεξεργαστών (αν υπάρχουν) είναι ακυρωμένα (Invalid). Η ιδιότητα αυτή επαληθεύτηκε χωρίς λάθη, άρα και η αμετάβλητη συνθήκη που αντιπροσωπεύει ισχύει στον αλγόριθμο. Τέλος επαληθεύτηκε η ιδιότητα για το αν τα κομμάτια μνήμης που αποστέλλονται σαν απάντηση σε κάποιο αίτημα είναι σε ορθή κατάσταση. Και αυτή η ιδιότητα επαληθεύτηκε χωρίς λάθη, και αυτό αποδεικνύει πως οι απαντήσεις σε αιτήματα δημιουργούνται μόνο με κομμάτια μνήμης των οποίων η κατάσταση είναι ορθή.

7.2 Μελλοντική δουλειά

Σε αυτή την διπλωματική εργασία έγινε μοντελοποίηση και επαλήθευση ορισμένων ιδιοτήτων του αλγορίθμου Greedy Order. Σαν μελλοντική δουλειά, θα μπορούσαν να ελεγχθούν και κάποιες άλλες ιδιότητες που αφορούν την ορθότητα αυτού του αλγορίθμου. Για παράδειγμα θα μπορούσε να ελεγχθεί η ιδιότητα για το αν ο αιτητής παραλαμβάνει πάντα τα δεδομένα που του στάλθηκαν σε κάποιο μήνυμα απάντησης που δημιουργήθηκε για κάποιο αίτημα του. Επίσης θα μπορούσε να ελεγχθεί η ιδιότητα για το όταν γίνεται εγγραφή σε κάποιο κομμάτι μνήμης, ότι μόνο ένας επεξεργαστής μπορεί να γράφει στο συγκεκριμένο κομμάτι μνήμης κάθε φορά. Άλλη μια ιδιότητα που θα μπορούσε να ελεγχθεί είναι ότι δεν μπορεί να δημιουργηθεί κάποιο μήνυμα απάντησης για κάποιο κομμάτι μνήμης αν δεν έχει προηγηθεί κάποιο αίτημα για αυτό. Ακόμη θα ήταν ενδιαφέρον να μοντελοποιηθεί και να επαληθευτεί ο αλγόριθμος Ring Order[15] ο οποίος συνδυάζει τα πλεονεκτήματα του αλγορίθμου Greedy Order που μοντελοποιήσαμε με τα πλεονεκτήματα του αλγορίθμου Ordering Point[15], και να γίνουν συγκρίσεις για τα μειονεκτήματα και πλεονεκτήματα του κάθε αλγορίθμου.

7.3 Επίλογος

Ο μοντελο-έλεγχος και η επαλήθευση συστημάτων είναι μια πολύ βοηθητική και επαναστατική μέθοδος, η οποία μπορεί να εξοικονομήσει τεράστια κεφάλαια από εταιρείες ανάπτυξης λογισμικού και όχι μόνο. Ιδιαίτερα με την συνεχή αύξηση της πολυπλοκότητας και των λειτουργιών που παρατηρείται με την πάροδο του χρόνου στα λογισμικά συστήματα, οι εταιρείες πρέπει να εκμεταλλευτούν στο έπακρο την ύπαρξη τέτοιων αυτοματοποιημένων

εργαλείων και τεχνικών, και να επενδύσουν κάποια κεφάλαια για την κατάρτιση προσωπικού, έτσι ώστε κάθε λογισμικό να ελέγχεται πρώτα ότι λειτουργεί ορθά και μετά να βγαίνει σε μαζική παραγωγή ή λειτουργία. Η αγνόηση ή η παράβλεψη τέτοιων εργαλείων και μεθόδων από εταιρείες, μπορεί να επιφέρουν τεράστιες οικονομικές συνέπειες που σε κάποιες περιπτώσεις μπορεί να αποδειχθούν και καταστροφικές.

Με την εμπειρία που αποκόμισα από την υλοποίηση και επαλήθευση του αλγορίθμου, και μέσα από τα λάθη που βρήκα και αντιμετώπισα, συνειδητοποίησα ότι αν προσπαθούσα να υλοποιήσω τον συγκεκριμένο αλγόριθμο απευθείας σε κάποια γλώσσα προγραμματισμού, θα έκανα λάθη που θα ήταν δύσκολο να εντοπιστούν με απλές δοκιμές (testing) κάποιων εκτελέσεων του προγράμματος. Όπως για παράδειγμα την απώλεια μηνυμάτων που εντόπισα μετά την ολοκλήρωση της πρώτης υλοποίησης, όπου το λάθος εντοπίστηκε μετά από επαλήθευση της συγκεκριμένης ιδιότητας χρονικής λογικής. Από αυτό συμπεραίνουμε ότι σε κάθε υλοποίηση οποιουδήποτε συστήματος, μπορεί να υπάρξουν παράμετροι, που από λάθος του προγραμματιστή να μην ληφθούν σωστά υπόψη, και σαν αποτέλεσμα να έχουμε την κακή λειτουργία του συστήματος. Αν το κομμάτι που υλοποίησα ήταν μέρος ενός μεγαλύτερου συστήματος μιας μεγάλης εταιρίας, και δεν εφαρμοζόταν μοντελο-έλεγχος, τότε πολύ πιθανό το ολοκληρωμένο σύστημα να μην λειτουργούσε σωστά, γεγονός που θα ανάγκαζε την εταιρία στη συνέχεια να ξοδέψει τεράστια ποσά για την αποσφαλμάτωση του. Αυτό αποδεικνύει την σημαντικότητα του μοντελο-ελέγχου, ειδικά σε μεγάλα και πολύπλοκα συστήματα, γιατί μέσω της εφαρμογής αυτής της τεχνικής μπορεί αποφευχθούν τέτοια λάθη, και επίσης ο προγραμματιστής μπορεί να βοηθηθεί και να καταλάβει καλύτερα τον τρόπο λειτουργίας του συστήματος.

Βιβλιογραφία:

[1] Φιλίππου Άννα, Σημειώσεις Μαθήματος ΕΠΛ 412 – Λογική στην Πληροφορική, Τμήμα Πληροφορικής Πανεπιστήμιο Κύπρου, http://www.cs.ucy.ac.cy/~annap/EPL412/notes.html

[2] Mordechai Ben-Ari. *Principles of the Spin Model Checker.* Springer, 2008.

[3] Gerard J. Holzmann. *The SPIN model checker.* Pearson Education, 2004.

[4] Tom Axford. *Concurrent Programming.* Wiley, 1989.

[5] A. Cimatti, F. Giunchiglia, G. Mongardi, D. Romano, F. Torielli, P. Traverso. Model Checking Safety Critical Software with SPIN: An application to a Railway Interlocking System"**.** In *Proceedings of the 17th International Conference on Computer Safety, Reliability and Security, volume 1516 of Lecture Notes in Computer Science, Springer, 1998.*

[6] P. Kars. "The application of Promela and Spin in the B.O.S. Project". In *Proceedings of a DIMACS Workshop, volume 32 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1997.*

[7] H. E. Jensen, K. G. Larsen, A. Skou. "Modeling and analysis of a Collision Avoidance Protocol using Spin and UPPAAL"**.** In *Proceedings of the 2nd SPIN Workshop on Algorithms, Applications, Tool Use, Theory, 1996.*

[8] E. Fersman, B. Jonsson. "Abstraction of Communication Channels in Promela: A Case study"**.** In *Proceedings of the 7th International SPIN Workshop, volume 1885 of Lecture Notes in Computer Science, pages 187-204,Springer, 2000.*

[9] T. R. Andel, A. Yasinsac. "Automated Evaluation of Secure Route Discovery in MANET Protocols". In *Proceedings of the 15th International SPIN workshop, volume 5156 of Lecture Notes in Computer Science, Springer, 2008.*

[10] J. Held, J. Bautista, S. Koehl. "From a Few Cores to many: A Tera-Scale Computing Research overview**".** In *Proceedings of the 45th Design Automation Conference, 2008.*

[11] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K.Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, K. Yelick. "The Landscape of Parallel Computing Research: A view from Berkeley". *Technical Report, Electrical Engineering and Computer Sciences University of California at Berkeley, Technical Report No. UCB/EECS-2006-183,2006*.

[12] "*Multi-core processors,* From Wikipedia, the free encyclopedia" <http://en.wikipedia.org/wiki/Multi-core_processor>

[13] **"***Dual-core processors,* From Maxi-pedia, free online encyclopedia" <http://www.maxi-pedia.com/dual+core+processor>

[14] M. R. Marty and M. D. Hill. "Coherence Ordering for Ring-based Chip Multiprocessors"**.** In Proceedings of the *39th Annual IEEE/ACM International Symposium on Micro-architecture (MICRO-39 2006*), *2006*

[15] M. R. Marty, 'Cache coherence techniques for multi-core processors'**.** *PhD thesis, University of Wisconsin – Madison, 2008*

Παράρτημα Α

A.1 Κώδικας Mutual Exclusion with Exchange Instructions

```
/-------- Mutual exclusion with exhange instructions --------/

int bolt;

proctype Process1(){

int x1 = 1;
int temp1;

      do
      ::    atomic{
                  temp1=x1;
                  x1=bolt;
                  bolt=temp1;
                  }

                  do
                  ::    (x1 == 1)->atomic{
                                    temp1=x1;
                                    x1=bolt;
                                    bolt=temp1;
                                    }
                  ::    (x1 != 1) ->break;
                  od;

            printf("Process %d is currently executing its critical
section\n",_pid);
            atomic{
                  temp1=x1;
                  x1=bolt;
                  bolt=temp1;
                  }
            printf("Process %d remainder...\n",_pid);
      od;
      }


proctype Process2(){

int x2 = 1;
int temp2;

      do
```

```
      ::    atomic{
                  temp2=x2;
                  x2=bolt;
                  bolt=temp2;
                  }
                  do
                  ::    (x2 == 1)->atomic{
                                    temp2=x2;
                                    x2=bolt;
                                    bolt=temp2;
                                    }
                  ::    (x2 != 1) ->break;
                  od;

            printf("Process %d is currently executing its critical
section\n",_pid);
            atomic{
                  temp2=x2;
                  x2=bolt;
                  bolt=temp2;
                  }
            printf("Process %d remainder...\n",_pid);
      od;
      }



init{
atomic{
      run Process1();
      run Process2();
      }
}
```

A.2 Κώδικας Mutual Exclusion with Lock Instructions

```
/*-------- Mutual exclusion with lock instructions --------*/

int bolt;

proctype Process1(){
      do
      ::
            do
            ::(bolt == 1)->skip;
            ::(bolt != 1) ->break;
            od;
            atomic{
                  bolt=1;
                  printf("Process %d is currently executing its critical
section\n",_pid);
            }
            bolt=0;
            printf("Process %d remainder...\n",_pid);
      od;
      }


proctype Process2(){
      do
      ::
            do
            ::(bolt == 1)->skip;
            ::(bolt != 1) ->break;
            od;
            atomic{
                  bolt=1;
                  printf("Process %d is currently executing its critical
section\n",_pid);
            }
            bolt=0;
            printf("Process %d remainder...\n",_pid);
      od;
}


init{
atomic{
      run Process1();
      run Process2();
      }
}
```

A.3 Κώδικας Mutual Exclusion with Test and Set Instructions

```
/*-------- Mutual exclusion using test and set instructions --------*/
int bolt;

proctype Process1(){

int x1;

      do
      ::    atomic{
                  x1=bolt;
                  bolt=1;
                  }
                  do
                  ::    (x1 == 1)->atomic{
                                    x1=bolt;
                                    bolt=1;
                                    }
                  ::    (x1 != 1) ->break;
                  od;

            printf("Process %d is currently executing its critical
section\n",_pid);
            bolt=0;
            printf("Process %d remainder...\n",_pid);
      od;
      }

proctype Process2(){

int x2;

      do
      ::    atomic{
                  x2=bolt;
                  bolt=1;
                  }

                  do
                  ::    (x2 == 1)->atomic{
                                    x2=bolt;
                                    bolt=1;
                                    }
                  ::    (x2 != 1) ->break;
                  od;

            printf("Process %d is currently executing its critical
section\n",_pid);
            bolt=0;
            printf("Process %d remainder...\n",_pid);
      od;
      }


init{
atomic{
      run Process1();
      run Process2();
      }
```

```
}
```

A.4 Κώδικας μοντελοποίησης αλγορίθμου Greedy Order (αρχείο Greed_alg.pml)

```
//GLOBAL VARIABLES DECLARATION
//Communication Channels between Processors
chan P1_P2Req_mess_chan = [0] of {bit,int,byte,int,int,bit,int}
//{messtype,requester,data, owner_id,send_data,response,response_data}
chan P2_P3Req_mess_chan = [0] of {bit,int,byte,int,int,bit,int}
chan P3_P1Req_mess_chan = [0] of {bit,int,byte,int,int,bit,int}

#define N 2                    //Cache size
mtype  = {O,S,M,E,I,NP};       //Possible States of a block

int P1id=1;                    //Processors ids
int P2id=2;
int P3id=3;

typedef Cache{
      mtype state;
      byte block;
      int data;
}
Cache P1[N];
Cache P2[N];
Cache P3[N];
byte MaxVar = 66; //Maximum value of variable for requests must be //65<=MaxVar<=90 -- valye of chars in Bytes

//Variables used to acknowledge that all processors have initialized //their caches and can send requests
bool P1OK = false;
bool P2OK = false;
bool P3OK = false;
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////
//                              P1 CACHE
////////////////////////////////////////////////////////////////////////////////////////////////////
active proctype Process1(){
int i=0;
byte block = 65;        //requesting block
int requester=0;        //requester id
byte dblock =0;         //requested block in incoming message
bit messType=0;         //0 for read 1 for write
bit messtype_out=0;     //requesting operation
int owner_id=0;         //block owner
int send_data = 0;      //owner sent data
bit foundblock=0;       //block is in Processors cache (when ==1)
bit response=0;         //message type: 0 for request 1 or response
int data;               //data of block
byte block_requested=0;
int position = 0;       //position to store data received in cache
bit send=1;             //when processor can send request
bit re_issue=0;         //when to force re-issue
bit mess_type_req;      //requesting operation
//initialize P1cache
byte A='A';

atomic{
      do
            ::i<1->P1[i].state = O;
                     P1[i].data = 1;
                     P1[i].block = A;
                     A=A+1;
                     i++;
            ::(i>=1)&&(i<N)->P1[i].block = 0;
                  i++;
            ::i==N->break;
      od;
      P1OK=true;
}


////////////////////////////////////////////////////////////////////////
```

```
begin1:
      if
      ::(P1OK && P2OK && P3OK)->
      do
       ::atomic{(send==1)->P1_P2Req_mess_chan!messtype_out,P1id,block,0,0,0,0;
                  send=0;}
                  mess_type_req=messtype_out;
                  block_requested=block;
                  if
                  ::block==MaxVar->block=65;
                  ::else->block++;
                  fi;
                  if
                  ::messtype_out==0->messtype_out=1;
                  ::else->messtype_out=0;
                  fi;




      ::P3_P1Req_mess_chan?messType,requester,dblock,owner_id,send_data,response,data->

            if
            ::response==0->
               if
               //READ REQUEST RECEIVED
               ::messType==0->if

                           ::(requester==P1id)->if
                                               ::(re_issue==1)->
                                                     block=dblock;
                                                     messtype_out=mess_type_req;
                                                     re_issue=0;
                                               ::else->if
                                                     ::(send_data==0)->
                                          printf("Block was not found in any cache-Retrieve from memory!!");
                                                     ::else->skip;
                                               fi;
```

```
            fi;
            send=1;


        ::(requester!=P1id&&send_data==0)->
            foundblock=0;     //RECEIVED MESSAGE FOR OTHER PROCESSOR REQUEST
            i=0;
            do
            ::if
              ::(P1[i].block == dblock)->
                 if
                 ::((P1[i].state==O)|| (P1[i].state==E) || (P1[i].state==M)||(P1[i].state==S))->
                                  P1[i].state=O;            //CHANGE BLOCK STATE
                                  send_data = 1;
                                  owner_id=P1id;
                                  foundblock=1;
               P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,1,P1[i].data;
                                  //SEND RESPONSE MESS WITH DATA
                                  goto begin1; //break
                                 ::else->break;
                               fi;

                    ::(P1[i].block != dblock)->if
                                      ::i<N-1->i++;          //INCREMENT INDEX OF ARRAY
                                      ::else->break;         //BLOCK NOT IN CACHE
                                    fi;
              fi;
            od;

            //FORWARD MESSAGE TO NEXT PROCESSOR

            if
            ::foundblock == 0->

P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,response,data;
            ::else->skip;
            fi;
      fi;
```

```
//WRITE REQUEST RECEIVED
::messType==1->if
              ::(requester==P1id)->if
                                  ::(re_issue==1)->
                                         block=dblock;
                                         messtype_out=mess_type_req;
                                         re_issue=0;
                                  ::else->if
                                         ::(send_data==0)->
                                   printf("Block was not found in any cache-Retrieve from memory!!");
                                         ::else->skip;
                                         fi;
                                  fi;
                            send=1;


             ::(requester!=P1id)->
                   if                          //Snoop request
                   ::(block_requested==dblock)&&(mess_type_req==0)->
                         do    //search for the block in the array
                         ::if
                         ::(P1[i].block == dblock)->
                                     P1[i].state=I;           //CHANGE BLOCK STATE
                                     break;
                         ::(P1[i].block != dblock)->if
                                           ::i<N-1->i++;     //INCREMENT INDEX OF ARRAY
                                           ::i==N-1->break;
                                            fi;
                         fi;
                         od;
                         re_issue=1;

      P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,response,data;
                   ::else->

                   //RECEIVED A WRITE REQUEST FOR OTHER PROCESSOR - CHECK AND FORWARD
                   foundblock=0;
                   i=0;
                   do                                    //SEARCH THE BLOCK IN CACHE
```

```
                    ::(P1[i].block == dblock)->
                         if
                         ::((P1[i].state==O)|| (P1[i].state==E) || (P1[i].state==M))->
                                                P1[i].state=I;       //CHANGE BLOCK STATE
                                                send_data = 1;
                                                owner_id=P1id;
                                                foundblock=1;
          P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,1,P1[i].data;
                                                //SEND RESPONSE MESS WITH DATA
                                                goto begin1;
                                                // break;

                         ::else->if
                                 ::(P1[i].state==S)->
                                        P1[i].state=I;
                                        send_data = 0;
                                        break;
                                 ::else->break;
                                 fi;
                          fi;

                    ::(P1[i].block != dblock)->if
                                        ::i<N-1->i++;             //INCREMENT INDEX OF ARRAY
                                        ::else->break;                  //BLOCK NOT IN CACHE
                                          fi;
               od;

               //FORWARD MESSAGE TO NEXT PROCESSOR
               if
               ::foundblock == 0->

 P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,response,data

               ::else->skip;
               fi;
             fi;
          fi;//requester!=P1id
fi;//end if response ==0
```

```
::response == 1->
        if
        //RECEIVED RESPONSE FOR A READ OPERATION
        ::(messType==0)->           //RESPONSE TO THIS PROCESSOR'S REQUEST - CACHE THE MESSAGE
            if
            ::requester==P1id->
            if
            ::(re_issue==1)->
                  block=dblock;
                  messtype_out=mess_type_req;
                  re_issue=0;
            ::else->
            foundblock=0;
            i=0;
            if
            :: position < N->  P1[position].state = O;                  //STORE IN NEXT BLOCK
                               P1[position].data = data;
                               P1[position].block = dblock;
                               position++;
            :: else->position=1;
                               P1[position].state = O;        //START STORING FROM FIRST POSITION
                               P1[position].data = data;
                               P1[position].block = dblock;
                               position++;
            fi;
            fi;


            //INITIALIZE  VARS TO AVOID CONFLICT WHEN CREATING A NEW REQUEST
              owner_id=0;
              send_data=0;
              send=1;
             ::else->   //requester!=P1id
             P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,response,data;
            //FORWARD MESSAGE
       fi; //messtype==0

      //RECEIVED RESPONSE FOR A WRITE OPERATION
      ::(messType==1)->
```

```
if
::requester==P1id->           //RESPONSE TO THIS PROCESSOR'S REQUEST
  if
  ::(re_issue==1)->
      block=dblock;
      messtype_out=mess_type_req;
      re_issue=0;
  ::else->
  if
 ::position < N -> P1[position].state = M;
            //CACHE FULL - STORE IN FIRST BLOCK (i==0)
                   P1[position].data = data;
                   P1[position].block = dblock;
                        if
                        ::(P1[i].data==1)->
                                  P1[i].data=0;
                        ::else->P1[i].data=1;
                        fi;                              //WRITE TO BLOCK
                   position++;
              ::else-> position=1;
                  P1[position].state = M;
      P1[position].data = data;
                  P1[position].block = dblock;
                        if
                        ::(P1[i].data==1)->
                                  P1[i].data=0;
            ::else->P1[i].data=1;
                        fi;                                    //WRITE TO BLOCK
                     position++;
fi;

fi;                //INITIALIZE  VARS TO AVOID CONFLICT WHEN CREATING A NEW REQUEST
  owner_id=0;
  send_data=0;
  send=1;
 ::else->//requester!=this processor
      if
      ::block_requested==dblock->
            re_issue=1;
```

```
                        ::else->skip;
                         fi;
                  //RECEIVED A WRITE REQUEST FOR OTHER PROCESSOR - INVALIDATE IF FOUND AND FORWARD
                        i=0;
                        do    //search for the block in the array
                        ::if
                              ::(P1[i].block == dblock)->
                                          P1[i].state=I;           //CHANGE BLOCK STATE
                                          break;
                              ::(P1[i].block != dblock)->if
                                                      ::i<N-1->i++;  //INCREMENT INDEX OF ARRAY
                                                      ::i==N-1->break;
                                                       fi;
                        fi;
                        od;
                        P1_P2Req_mess_chan!messType,requester,dblock,owner_id,send_data,response,data;
            //FORWARD MESSAGE TO NEXT PROCESSOR
            fi;//messtype==1
      fi;//end if response ==1
fi;

od;
::else goto begin1;
fi;

}
```