Ατομική Διπλωματική Εργασία

ΠΡΟΔΙΑΓΡΑΦΗ ΚΑΙ ΠΡΟΣΟΜΟΙΩΣΗ ΧΡΟΝΙΣΜΕΝΩΝ ΚΑΤΑΝΕΜΗΜΕΝΩΝ ΑΛΓΟΡΙΘΜΩΝ ΑΜΟΙΒΑΙΟΥ ΑΠΟΚΛΕΙΣΜΟΥ ΧΡΗΣΙΜΟΠΟΙΩΝΤΑΣ ΤΟ ΕΡΓΑΛΕΙΟ TEMPO

ΜΑΡΙΟΣ ΠΑΠΑΧΡΙΣΤΟΔΟΥΛΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Μάιος 2012

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Προδιαγραφή και Προσομοίωση Χρονισμένων Κατανεμημένων Αλγορίθμων Αμοιβαίου Αποκλεισμού Χρησιμοποιώντας το Εργαλείο TEMPO

Μάριος Παπαχριστοδούλου

Επιβλέπων Καθηγητής
Χρύσης Γεωργίου

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

Μάιος 2012

Ευχαριστίες

Θα ήθελα να εκφράσω ένα πολύ μεγάλο ευχαριστώ στον επιβλέποντα καθηγητή μου, κύριο Χρύση Γεωργίου, για την πολυτιμότατη και συνεχή βοήθεια που μου προσέφερε. Οι γνώσεις του και η βοήθεια του μου ήταν απαραίτητα στοιχεία για να φτάσω στο σημείο αυτό. Τον ευχαριστώ από τα βάθη της ψυχής μου για τη βοήθεια του και του εύχομαι κάθε επιτυχία και ευτυχία στο υπόλοιπο της ζωής του. Επίσης, ένα μεγάλο ευχαριστώ οφείλω στον Χρίστο Πλουτάρχου για την σημαντικότατη βοήθεια του σε καίρια σημεία που με δυσκόλεψαν. Οι επεμβάσεις του ήταν καθοριστικές στην ολοκλήρωση αυτής της διπλωματικής εργασίας, και αναγνωρίζω επίσης τον πολύτιμο χρόνο που μου θυσίασε ενώ παράλληλα προετοιμαζόταν να παντρευτεί. Του εύχομαι βίο ανθόσπαρτο και πολλούς απογόνους.

Τέλος, ένα μεγάλο ευχαριστώ οφείλω στην οικογένεια μου, η οποία με στήριξε σε μια δύσκολη περίοδο της ζωής μου, και για την υποστήριξη τους κατά την συνολική διάρκεια των σπουδών μου.

Περίληψη

Στο πλαίσιο της ΑΔΕ μου κλήθηκα να δημιουργήσω μια προδιαγραφή για ασύγχρονους κατανεμημένους αλγόριθμους αμοιβαίου αποκλεισμού σε γλώσσα TIOA με τη χρήση του μαθηματικού μοντέλου TIOA (Timed Input / Output Automata). Η προδιαγραφή των αλγορίθμων έγινε με τη χρήση του εργαλείου Tempo toolkit. Οι αλγόριθμοι που προδιαγράφηκαν αποτελούν διαφορετικές προσεγγίσεις για λύση του ιδίου προβλήματος: του αμοιβαίου αποκλεισμού σε κατανεμημένα συστήματα. Το πρόβλημα του αμοιβαίου αποκλεισμού παρουσιάζεται όταν περισσότερες από μια διεργασίες προσπαθούν ταυτόχρονα να αποκτήσουν πρόσβαση στον ίδιο (κοινό) πόρο, είτε αυτός είναι κοινή μνήμη είτε κάποιος άλλος πόρος σε ένα δίκτυο. Επειδή στα κατανεμημένα συστήματα δεν μπορεί να υπάρξει κάποιος συντονιστής (αφού οι επιμέρους διεργασίες είναι αυθαίρετες και επιρρεπείς σε σφάλματα), χρειάζεται οι διεργασίες να συνεργάζονται με ανταλλαγή μηνυμάτων και η κάθε μια από αυτές να αποφασίζει τοπικά αν πρέπει ή όχι να αποκτήσει πρόσβαση στον κοινό πόρο. Το γεγονός αυτό αποτελεί πρόκληση για ένα προγραμματιστή, μιας και ο προγραμματισμός κατανεμημένων αλγόριθμων είναι πολύ δύσκολος. Πρέπει να λαμβάνονται υπόψη όλες οι πιθανές περιπτώσεις που μπορεί να προκύψουν κατά τη διάρκεια μιας προσομοίωσης.

Αφού ολοκλήρωσα την προδιαγραφή των αλγορίθμων σε μοντέλο TIOA, προχώρησα στη μετάφραση τους σε κώδικα Java μέσω ενός πειραματικού εργαλείου, του tempo2java plugin για το εργαλείο Tempo. Δυστυχώς, το εργαλείο δεν βρίσκεται ακόμη σε ολοκληρωμένο στάδιο, γεγονός το οποίο έκανε τη δουλειά μου πιο δύσκολη.

Στη συνέχεια χρησιμοποίησα το έτοιμο εργαλείο tempo2java για να μεταφραστεί ο κώδικας μου αυτοματοποιημένα σε πηγαίο κώδικα Java. Αφού μετάφρασα τον κώδικα μου σε Java χρησιμοποιώντας το εργαλείο αυτό, προχώρησα σε πειραματική δοκιμή του τρέχοντας κάποια πειραματικά σενάρια τοπικά στον υπολογιστή μου. Αυτό το έπραξα με τη χρήση της βιβλιοθήκης "MPJ Express", η οποία υλοποιεί μια διεπαφή ανταλλαγής μηνυμάτων (Message Passing Interface - MPI) στη γλώσσα Java.

Ολοκληρώνοντας την ΑΔΕ αυτή έμαθα πάρα πολλά για το εργαλείο Tempo και τον κατανεμημένο προγραμματισμό. Επίσης, είχα την δυνατότητα να έχω τριβή με την βιβλιοθήκη MPJ Express, την οποία χρησιμοποίησα για να δημιουργήσω πολλαπλά νήματα στη μηχανή μου και να τρέξω τα πειράματα μου.

Περιεχόμενα

Κεφάλαιο 1

Εισαγωγή

1.1 Κίνητρο

Στην ενότητα αυτή περιγράφεται το κίνητρο που αποτέλεσε το έναυσμα για την δημιουργία της διπλωματικής αυτής εργασίας, το οποίο ξεκίνησε από το ενδιαφέρον που έχει ο προγραμματισμός κατανεμημένων αλγορίθμων [1]. Κατανεμημένος αλγόριθμος είναι ο αλγόριθμος που προσπαθεί να επιλύσει ένα πρόβλημα μέσω κατανεμημένου υπολογισμού. Στον κατανεμημένο υπολογισμό δεν υπάρχει κάποιος κεντρικός πυρήνας που να μπορεί να πάρει όλες τις αποφάσεις ή να συντονίσει όλες τις υπόλοιπες διεργασίες. Αντίθετα, υπάρχουν πολλοί κόμβοι ή διεργασίες διασυνδεδεμένοι μέσω κάποιας μορφής δικτύου (που πιθανό να βρίσκονται φυσικά διεσπαρμένοι σε διαφορετικές γεωγραφικές περιοχές), οι οποίοι τρέχουν παράλληλα και επικοινωνούν ανταλλάζοντας μηνύματα, ενώ οι αποφάσεις παίρνονται τοπικά. Σκοπός των κατανεμημένων αλγορίθμων είναι η σωστή συνεργασία μεταξύ αυτών των κόμβων / διεργασιών για την ορθή λύση κάποιου προβλήματος. Ένας κατανεμημένος αλγόριθμος χαρακτηρίζεται ασύγχρονος όταν οι κόμβοι του δικτύου μπορούν να εκτελούν τις ενέργειες τους χωρίς συγχρονισμό (δηλαδή αυθαίρετα). Το γεγονός αυτό κάνει τη δουλειά του προγραμματιστή κατανεμημένων αλγόριθμων πολύ δύσκολη. Χωρίς σωστό σχεδιασμό και ιδανική οργάνωση του προβλήματος, υπάρχει περίπτωση να μας διαφύγουν κάποιες περιπτώσεις οι οποίες να προκαλέσουν πρόβλημα στην ομαλή εκτέλεση ενός καταμερισμένου αλγόριθμου. Το μεγαλύτερο από τα προβλήματα που παρουσιάζονται είναι

οι περιπτώσεις κατάρρευσης κόμβων, οι οποίες αν δεν υπολογιστούν σωστά μπορούν να προκαλέσουν κατάρρευση ολόκληρου του συστήματος.

Το πρόβλημα του Αμοιβαίου Αποκλεισμού [2] αποτελεί ένα από τα πιο αρχέγονα προβλήματα της επιστήμης Πληροφορικής, ιδιαίτερα με την ανάπτυξη των λειτουργικών συστημάτων. Παρουσιάζεται σε συστήματα όπου διάφορες διεργασίες προσπαθούν ταυτόχρονα να χρησιμοποιήσουν κάποιο κοινό πόρο, είτε αυτός αποτελεί συγκεκριμένη θέση μνήμης είτε κάποια φυσική μονάδα ενός δικτύου. Ιδεατά, μόνο μια διεργασία πρέπει να έχει πρόσβαση στην κρίσιμη αυτή περιοχή (δηλαδή τον κοινό πόρο). Σε τοπικά συστήματα (ενός επεξεργαστή) η λύση του προβλήματος επέρχεται με τη χρήση σημαφόρων. Σε κατανεμημένους ωστόσο αλγόριθμους δεν υπάρχει κάποιος συντονιστής. Αντίθετα, οι κόμβοι πρέπει να συνεργαστούν σωστά και μέσω ανταλλαγής μηνυμάτων η καθεμία να αποφασίζει τοπικά αν πρέπει ή όχι να μπει στην κρίσιμη περιοχή.

Για τον σχεδιασμό αλγορίθμων αμοιβαίου αποκλεισμού σε κατανεμημένα συστήματα είναι κρίσιμο να προσέξουμε τις περιπτώσεις κατάρρευσης κόμβων / απώλειας μηνυμάτων, τα οποία μπορεί να προκαλέσουν ανωμαλίες στην λειτουργία εκτέλεσης του αλγορίθμου. Για παράδειγμα, αποτυχία κόμβου που κρατά μια σκυτάλη πιθανό να προκαλέσει παρατεταμένη στέρηση στην είσοδο στην κρίσιμη περιοχή. Για το λόγο αυτό η υλοποίηση τους αποτελεί πρόκληση για ένα προγραμματιστή.

Για την υλοποίηση κατανεμημένων αλγόριθμων χρησιμοποιείται συχνά το μαθηματικό μοντέλο Χρονισμένων Αυτομάτων Εισόδου / Εξόδου (Timed Input / Output Automata - TIOA) [1] που είναι κατάλληλο για μοντελοποίηση τέτοιων αλγορίθμων και υλοποίηση τους στο ψηλότερο σημείο αφαιρετικότητας. Το εργαλείο Tempo [3], το οποίο προσφέρεται δωρεάν από τον οργανισμό Veromodo Inc. [7], χρησιμοποιεί το μοντέλο αυτό για την υλοποίηση, σχεδιασμό και προσομοίωση κατανεμημένων αλγορίθμων σε γλώσσα TIOA.

1.2 Σκοπός

Ο σκοπός της διπλωματικής αυτής εργασίας είναι η προδιαγραφή ασύγχρονων κατανεμημένων αλγορίθμων με τη χρήση του εργαλείου Tempo toolkit, η μετάφραση τους σε java χρησιμοποιώντας τον tempo2java μεταφραστή [5] και η πειραματική τους αξιολόγηση τρέχοντας σενάρια προσομοίωσης [8]. Συγκεκριμένα, οι αλγόριθμοι υλοποιούν

Αμοιβαίο Αποκλεισμό σε πλήρως συνδεδεμένα κατανεμημένα συστήματα. Η εργασία αυτή βασίζεται στην αρχική προσέγγιση της γλώσσας Tempo του Στέφανου Ατριαν Ματεϊ [4], στην εργασία του Χρίστου Πλουτάρχου [6], στην εργασία του Peter M. Musial ο οποίος υλοποίησε το εργαλείο tempo2java [5], καθώς και στο βιβλίο [2], από το οποίο αντλήθηκαν οι πληροφορίες για τους αλγόριθμους που υλοποιήθηκαν.

1.3 Μεθοδολογία

Αρχικά, χρειάστηκε να εκπαιδευτώ για την χρήση του εργαλείου Tempo. Για να γίνει αυτό μελετήθηκαν διάφορα κείμενα για τα μοντέλα IOA και TIOA και για τις διαφορές τους. Στη συνέχεια εγκατέστησα το πακέτο λογισμικού Tempo toolkit που αναπτύχθηκε από την Veromodo Inc. [7] και επικεντρώθηκα στην ανάπτυξη, διόρθωση και προσομοίωση απλών αλγορίθμων για εκμάθηση του εργαλείου. Ακολούθως, μελέτησα του αλγόριθμους προς υλοποίηση [2] μέχρι να τους κατανοήσω πλήρως. Με την πάροδο του χρόνου και αφού έγιναν αρκετές αλλαγές, οι αλγόριθμοι αυτοί υλοποιήθηκαν πλήρως στο εργαλείο Tempo. Τελικά, οι αλγόριθμοι μεταφράστηκαν σε πηγαίο κώδικα Java, γεγονός το οποίο μας επέτρεψε να δημιουργήσουμε σενάρια προσομοιώσεων χρησιμοποιώντας της βιβλιοθήκη "MPJ Express" [8], η οποία μας επιτρέπει να δημιουργήσουμε πολλές διεργασίες που να επικοινωνούν με ανταλλαγή μηνυμάτων . Οι αλγόριθμοι αξιολογήθηκαν ως προς τον αριθμό των μηνυμάτων που αποστέλλονταν καθώς και ως προς την παρατεταμένη στέρηση κόμβων από την κρίσιμη περιοχή.

1.4 Δομή Ατομικής Διπλωματικής Εργασίας

Στο κεφάλαιο 2 περιγράφεται αναλυτικά η γλώσσα TIOA, αναφέρονται οι διαφορές της με την γλώσσα IOA και επιδεικνύεται το εργαλείο Tempo toolkit μέσω περιγραφικής ανάλυσης και παραδειγμάτων. Στο κεφάλαιο 3 περιγράφονται και επεξηγούνται αναλυτικά οι δυο αλγόριθμοι που υλοποιήθηκαν για την αντιμετώπιση του προβλήματος του Αμοιβαίου Αποκλεισμού σε κατανεμημένα συστήματα. Στο κεφάλαιο 4 αναλύεται η υλοποίηση των αλγορίθμων αυτών σε γλώσσα TIOA με τη χρήση του εργαλείου Tempo. Στο κεφάλαιο 5 περιγράφεται η πειραματική αξιολόγηση των αλγορίθμων τρέχοντας σενάρια προσομοίωσης με χρήση της βιβλιοθήκης "MPJ Express". Τέλος, στο κεφάλαιο 6 αναφέρονται τα συμπεράσματα που εξάγονται από την διπλωματική αυτή εργασία, τα προβλήματα που αντιμετωπίστηκαν και πιθανή μελλοντική εργασία που πηγάζει από αυτή την ατομική διπλωματική εργασία.

Κεφάλαιο 2

Το εργαλείο Tempo - TIOA toolkit

2.1 Μοντέλο Αυτομάτων Εισόδου / Εξόδου

Το μοντέλο Αυτομάτων Εισόδου / Εξόδου (Input-Output Automata ή για συντομία IOA) [1] αποτελεί μια αξιόλογη μαθηματική βάση, με τη χρήση της οποίας μπορεί ένας αλγόριθμος να περιγραφεί και να υλοποιηθεί σε διάφορα επίπεδα αφαιρετικότητας. Συγκεκριμένα, το μοντέλο IOA μας επιτρέπει να εκφράσουμε ένα αλγόριθμο στο πιο ψηλό επίπεδο αφαιρετικότητας, πράγμα το οποίο παρέχει μεγάλες ευκολίες στην προσομοίωση ή και επιβεβαίωση κάποιων αλγορίθμων.
Ένα IOA αποτελείται από ένα σύνολο ενεργειών (actions), ένα σύνολο καταστάσεων (states), ένα σύνολο μεταβάσεων (transitions) και ένα σύνολο εργασιών (tasks). Οι ενέργειες που αποτελούν ένα IOA χωρίζονται σε τρεις υποκατηγορίες :

- Εσωτερικές ενέργειες (internal)
- Ενέργειες εισόδου (input)
- Ενέργειες εξόδου (output)

Οι 2 τελευταίες κατατάσσονται και ως εξωτερικές ενέργειες, και είναι αυτές που χρησιμοποιούνται για να παρέχουν επικοινωνία μεταξύ των αυτομάτων.

Δομικά, ένα αυτόματο IOA (ας το ονομάσουμε αυτόματο Α)αποτελείται από τα εξής μέρη:

- Την Υπογραφή (Signature) που συμβολίζεται ως sig(A) και περιλαμβάνει το σύνολο όλων των ειδών ενεργειών του Α.
- Το σύνολο καταστάσεων (States) που συμβολίζονται ως states(A) και περιλαμβάνει όλες τις μεταβλητές καταστάσεις στις οποίες πιθανό να βρίσκεται το αυτόματο.
- Το σύνολο αρχικών καταστάσεων (Start States) που συμβολίζεται ως start(A) και είναι ένα μη κενό υποσύνολο του συνόλου καταστάσεων.
- Το σύνολο σχέσεων μεταβάσεως (transition relations) το οποίο αποτελείται από μεταβάσεις ή όπως αλλιώς καλούνται, βήματα (τριάδες της μορφής state-action-state).
- Το σύνολο εργασιών (tasks), που είναι ένα σύνολο ενεργειών οι οποίες ελέγχονται τοπικά. Μπορεί δηλαδή να περιλαμβάνει ενέργειες εξόδου ή εσωτερικές ενέργειες, αλλά όχι ενέργειες εισόδου.

Μια ενέργεια ε του αυτομάτου Α θεωρείται ενεργή (enabled) σε μια κατάσταση s αν υπάρχει μια άλλη κατάσταση s' τέτοια ώστε η τριάδα/βήμα s-ε-s' ανήκει στο σύνολο μεταβάσεων του αυτομάτου Α. Εξαίρεση αποτελούν οι ενέργειες εισόδου, οι οποίες είναι ενεργές σε όλες τις καταστάσεις. Αυτό σημαίνει πως το αυτόματο δεν μπορεί να σταματήσει τη λειτουργία μιας ενέργειας εισόδου. Αντίθετα, οι εσωτερικές ενέργειες, καθώς και οι ενέργειες εξόδου, μπορούν να έχουν προϋποθέσεις (τις οποίες καλούμε "preconditions") οι οποίες καθορίζουν πότε οι ενέργειες αυτές είναι ενεργές. Η εξωτερικά ορατή συμπεριφορά ενός αυτομάτου Α καλείται ίχνος (trace). Αποτελείται από μια σειρά εναλλασσόμενων εξωτερικών ενεργειών.

Τα IOA επιτρέπουν την σύνθεση (composition) διάφορων αυτομάτων για τη δημιουργία ενός συστήματος. Στην σύνθεση αυτομάτων, κάθε ενέργεια εξόδου ε ενός αυτομάτου Α ταυτίζεται με την ενέργεια εισόδου ε άλλων αυτομάτων Α' (προσοχή: είναι σημαντικό το όνομα των 2 αυτών ενεργειών να είναι το ίδιο). Σε αυτή την περίπτωση, μόλις το αυτόματο Α εκτελέσει την ενέργεια εξόδου ε, όλα τα άλλα αυτόματα που έχει την ε σαν ενέργεια εισόδου την εκτελεί παράλληλα. Με αυτό τον τρόπο επιτυγχάνεται η επικοινωνία μεταξύ αυτομάτων.

2.2 Μοντέλο Χρονισμένων Αυτομάτων Εισόδου / Εξόδου

Εξέλιξη του μοντέλου Αυτομάτων Εισόδου/Εξόδου (IOA) που περιγράφεται στην προηγούμενη ενότητα αποτελεί το μοντέλο Χρονισμένων Αυτομάτων Εισόδου/Εξόδου (Timed Input/Output Automata - TIOA), ένα μοντέλο στο οποίο οι βασικές δομικές μονάδες

είναι μηχανές καταστάσεων (state machines) [1]. Η κύρια διαφορά των 2 είναι πως στο μοντέλο TIOA δεν περιοριζόμαστε μόνο σε διακριτά βήματα, αλλά μπορεί να έχουμε και συνεχή εξέλιξη, η οποία υλοποιείται με τη χρήση τροχιών (trajectories). Στην ουσία, ένα αυτόματο TIOA χωρίς τροχιές είναι πρακτικά ένα αυτόματο IOA. Ένα αυτόματο TIOA αποτελείται από τα εξής μέρη:

- Ένα σύνολο X από εξωτερικές μεταβλητές.
- Ένα σύνολο S από καταστάσεις.
- Ένα μη κενό σύνολο Θ απο αρχικές καταστάσεις, το οποίο είναι προφανώς υποσύνολο του S.
- Ένα σύνολο E από εξωτερικές ενέργειες και ένα σύνολο I από εσωτερικές ενέργειες. Τα 2 σύνολα αυτά **δεν** πρέπει να περιέχουν κοινά στοιχεία.
- Ένα σύνολο D από διακριτές μεταβάσεις (του τύπου state-action-state).
- Ένα σύνολο T από τροχιές.

Όπως είναι προφανές, οι τροχιές εισάγουν την έννοια του χρόνου στα αυτόματα TIOA, διαφοροποιώντας τα από τα IOA αυτόματα. Οι τιμές των στοιχείων σε μια κατάσταση ενός IOA αλλάζουν με την πάροδο του χρόνου μόνο ως αποτέλεσμα των διακριτών μεταβάσεων. Στην περίπτωση ενός αυτόματου TIOA οι τιμές μπορούν να αλλάξουν και στο χρόνο μεταξύ των διακριτών μεταβάσεων λόγω εξωτερικών επιρροών.

Όπως και στα IOA, στα TIOA επιτρέπεται η σύνθεση αυτομάτων για να παράξουν ένα πρότυπο ενός μεγαλύτερου συστήματος. Όλα τα αυτόματα TIOA μιας σύνθεσης συμμετέχουν στο σύνολο των τροχιών T ταυτόχρονα, επιτρέποντας να περάσει (ιδεατά) το ίδιο χρονικό διάστημα. Όπως και στα IOA, κάθε ενέργεια εξόδου ε, όταν εκτελεστεί, προκαλεί την ταυτόχρονη εκτέλεση των αντίστοιχων ενεργειών εισόδου ε στα υπόλοιπα αυτόματα της σύνθεσης.

2.3 Διαφορές IOA - TIOA

Συνοπτικά, οι διαφορές των δυο αυτών γλωσσών είναι οι ακόλουθες [4]:

- Η γλώσσα TIOA περιλαμβάνει επιπρόσθετα στοιχεία για την υλοποίηση του χρόνου, όπως είναι οι συνεχείς τύποι δεδομένων, οι τροχιές και οι σχέσεις παύσης για τις μεταβάσεις.

- Το TIOA περιέχει τους πόρους για τον καθορισμό των συναρτήσεων και των παραγόμενων μεταβλητών.
- Το TIOA περιέχει δυνατότητες για Nondeterministic Resolution (NDR) κατασκευαστές, οι οποίοι χρησιμοποιούνται για να παρέχουν πληροφορίες σχεδιασμού (scheduling information) στον προσομοιωτή TIOA.
- Το TIOA περιέχει πόρους για την καθοδήγηση της προσομοίωσης των τροχιών.
- Στο TIOA, σημεία για χειριστές σε τύπους δεδομένων (κάτι αντίστοιχο με τα typedefs και τα structs στην γλώσσα C) καθορίζονται χρησιμοποιώντας ένα κατασκευαστή λεξιλογίου (vocabulary). Στο IOA τα σημεία αυτά καθορίζονται σε βοηθητικά αρχεία. Η μέθοδος που χρησιμοποιείται στα TIOA έχει τα εξής πλεονεκτήματα:

    1. Μας δίνεται η δυνατότητα υλοποίησης ενός συστήματος TIOA σε ένα μόνο αρχείο, χωρίς να υπάρχει η ανάγκη για καταμερισμό του κώδικα σε βοηθητικά αρχεία (κάτι το οποίο είναι ωστόσο πραγματοποιήσιμο).
    2. Το TIOA είναι πιο υποκείμενο από το IOA λόγω της χρήσης των αποδεικτών θεωρήματος που εμπεριέχει.
    3. Οι ορισμοί των αυτομάτων και των λεξιλογίων χρησιμοποιούν τον ίδιο μηχανισμό παραμετροποίησης, ενώ στο IOA οι βοηθητικές προδιαγραφές χρησιμοποιούσαν διαφορετικό μηχανισμό από εκείνο για τον ορισμό των αυτομάτων.

2.4 Το Εργαλείο TEMPO και η Γλώσσα TIOA

Στην ενότητα αυτή περιγράφονται οι προδιαγραφές / δυνατότητες του εργαλείου TEMPO και της γλώσσας TIOA που χρησιμοποιεί. Το εργαλείο Tempo toolkit αναπτύχθηκε από την Veromodo Inc (που αποτελείται από ερευνητές του Πανεπιστημίου του Connecticut και του Massachusetts Institute of Technology) και διανέμεται δωρεάν [7]. Παρέχεται σε eclipse-based διαφάνεια χρηστών. Πριν να προχωρήσω σε περιγραφή της γλώσσας θα ήθελα να αναφέρω ότι το TEMPO - TIOA toolkit αποτελείται από τα ακόλουθα εργαλεία:

- Τον ελεγκτή (Syntax and Semantic Checker) ο οποίος χρησιμοποιείται για να ελέγξει ότι οι προδιαγραφές του αλγόριθμου είναι συντακτικά και σημαντολογικά ορθοί.
- Τον προσομοιωτή, ο οποίος χρησιμοποιείται για την προσομοίωση της εκτέλεσης του συστήματος σύμφωνα με το πρόγραμμα (scheduler). Βρίσκεται ακόμα σε εξελίξιμο στάδιο και βελτιώνεται συνεχώς.
- Το εργαλείο απόδειξης (Theorem Prover Tool - PVS), το οποίο χρησιμοποιείται για τον έλεγχο ορθότητας του αλγορίθμου.

- Τον μεταφραστή Tempo2Java ο οποίος παίρνει ως είσοδο μια προδιαγραφή σε TIOA γλώσσα και την μετατρέπει σε εκτελέσιμο κώδικα Java. Όπως και ο προσομοιωτής, βρίσκεται ακόμα σε πρώιμο στάδιο (alpha testing).

Για την πραγμάτωση της διπλωματικής αυτής εργασίας χρησιμοποιήθηκε ο ελεγκτής και ο προσομοιωτής (μόνο στο στάδιο της εκμάθησης το υ εργαλείου) ενώ για την αξιολόγηση των αλγορίθμων χρησιμοποιήθηκε ο μεταφραστής Tempo2Java.

2.4.1 Γλώσσα TIOA

Στην ενότητα αυτή περιγράφονται όλα τα δομικά στοιχεία που είναι απαραίτητα για να εκφράσουμε ένα αλγόριθμο σε γλώσσα TIOA [3].

2.4.1.1 Τύποι δεδομένων

Η TIOA επιτρέπει στους χρήστες να καθορίσουν τις ενέργειες και τις καταστάσεις των αυτομάτων αφαιρετικά, χρησιμοποιώντας μαθηματικές εκφράσεις χωρίς να πρέπει να παρασχεθούν οι αντιπροσωπεύσεις για αυτές τις αφαιρετικότητες. Περιληπτικά οι τύποι δεδομένων που δίνονται στην γλώσσα TIOA είναι οι ακόλουθοι:

- **Boolean:** Ο βασικότερος αρχέγονος τύπος δεδομένων. Μπορεί να περιέχει 2 τιμές, την 0 (false) και 1 (true). Εκτός από τις βασικές πράξεις σύγκρισης που υποστηρίζονται σε μια boolean έκφραση (> , < , = , $\vee$, $\wedge$, $\neg$), στην γλώσσα TIOA μπορούμε επίσης να χρησιμοποιήσουμε υπαρξιακούς ποσοδείκτες ($\exists$ και $\forall$), συνεπαγωγή (=>) και λογική ισοδυναμία (<=>).
- **Nat:** Φυσικός αριθμός. Το σύνολο των φυσικών αριθμών περιλαμβάνει όλους τους μη αρνητικούς ακέραιους αριθμούς. Υποστηρίζουν τις έτοιμες συναρτήσεις succ(x), pred(x), min(x,y), max(x,y), div(x,y) και mod(x,y).
- **Int:** Ακέραιος αριθμός. Υποστηρίζει όλες τις συναρτήσεις των Nat και την επιπρόσθετη έτοιμη συνάρτηση abs(x).
- **Real:** Πραγματικός αριθμός. Υποστηρίζει τις συναρτήσεις min(x,y), max(x,y), abs(x) και floor(x).
- **DiscreteReal:** Είναι και πάλι πραγματικός αριθμός, με τη διαφορά ότι δεν αλλάζουν μεταξύ των διακριτών βημάτων. Είναι ιδεατός τύπος για να οριστεί το φυσικό ρολόι ενός κόμβου (αυτομάτου).

- **AugmentedReal:** Επεκτείνει το σύνολο των πραγματικών αριθμών (Real) προσθέτοντας τις τιμές $-\infty$ και $+\infty$.
- **Char:** Χαρακτήρας. Επιτρέπει την σύγκριση (για παράδειγμα η συνθήκη 'B' > 'A' επιτρέπεται και είναι αληθής, μιας και η σύγκριση γίνεται στην τιμή του συγκεκριμένου χαρακτήρα στον πίνακα ASCII).
- **String:** Μια ακολουθία από χαρακτήρες. Μπορεί να δηλωθεί και ως Seq[Char] που θα επεξηγηθεί πιο κάτω.
- **Array:** Πίνακας. Για κάθε n>0, η δήλωση Array[T1, ...,Tn, E] αποτελεί ένα n-διάστατο πίνακα στοιχείων τύπου E που συντάσσονται από τα στοιχεία των τύπων T1,...,Tn. Για παράδειγμα, ένας δυσδιάστατος πίνακας char_array τύπου Char ορίζεται ως char_array:Array[Int,Int,Char].
- **Set:** Ένα σύνολο. Τα στοιχεία του Set[t] είναι πεπερασμένα σύνολα στοιχείων του τύπου t. Η TIOA υποστηρίζει σχεδόν όλες τις πράξεις συνόλων: ένωση ($\cup$) , τομή ($\cap$), ανήκει ($\in$), δεν ανήκει ($\notin$), υποσύνολο ($\subset$), υποσύνολο ή ίσο ν ($\subseteq$), μη υποσύνολο ($\not\subset$), ισότητα (=), διαφορά ή σχετικό συμπλήρωμα (-) και κενό σύνολο (Ø). Επίσης υποστηρίζει έτοιμες συναρτήσεις όπως size(S), η οποία επιστρέφει το πλήθος των στοιχείων στο σύνολο, και insert(e,S) ή delete(e,S) οι οποίες προσθέτουν και διαγράφουν το στοιχείο e από το σύνολο S αντίστοιχα.
- **Seq:** Αντιπροσωπεύει μια ουρά. Τα στοιχεία του Seq[t] είναι πεπερασμένες ουρές στοιχείων του τύπου t. Η TIOA υποστηρίζει κάποιες έτοιμες συναρτήσεις για τις ουρές. Περιληπτικά, αυτές είναι: insert element (|-), remove element (-|), element is in ($\in$), element is not in ($\notin$) , head(S) για το πρώτο στοιχείο που μπήκε στην ουρά, tail(S) για το τελευταίο στοιχείο που μπήκε στην ουρά, len(S) για το μήκος της ουράς και S[n] για το n-οστό στοιχείο της ουράς.

2.4.1.2 Επιτρεπόμενες Εκφράσεις

Παρακάτω ακολουθεί μια σύντομη περιγραφή των διαφόρων επιτρεπόμενων εκφράσεων στη γλώσσα TIOA.

- **Enumeration:** Απαρίθμηση. Αντίστοιχα με το enumeration στην γλώσσα C, η έκφραση αυτή μας επιτρέπει να δημιουργήσουμε ένα σύνολο με στοιχεία και να το ορίσουμε ως δικό μας τύπο δεδομένων. Οι απαριθμήσεις πρέπει να ορίζονται στην κορυφή του αρχείου όπου βρίσκεται ο κώδικας ή σε βοηθητικά αρχεία (vocabulary files - κάτι αντίστοιχο με τα header files στην γλώσσα C). Παράδειγμα χρήσης:

types Process: Enumeration [P1,P2,P3,P4]

Το πιο πάνω μας επιτρέπει τώρα να δηλώσουμε μεταβλητές του τύπου Process, και οι οποίες μπορούν να πάρουν τιμή P1, P2, P3 ή P4. Ισχύει η σειρά με την οποία δηλώνονται οι επιτρεπόμενες τιμές, δηλαδή αν έχω μια μεταβλητή temp η οποία είναι τύπου process και έχει τιμή P2, δηλώνοντας "temp:=temp+1;" η μεταβλητή temp θα πάρει την τιμή P3.

- **Where:** Όπου. Η γλώσσα TIOA μας επιτρέπει να χρησιμοποιήσουμε ένα "where" για να περιορίσουμε το πεδίο τιμών που μπορεί να πάρει μια μεταβλητή (χρησιμοποιείται συνήθως σε δηλώσεις ενεργειών / αυτομάτων για να περιορίσουμε τις επιτρεπτές τιμές που μπορούν να πάρουν οι παράμετροι). Παράδειγμα χρήσης:

automaton Process (processID : Nat) where processID>0

- **Choose:** Η εντολή αυτή μας επιτρέπει να παίρνουμε τυχαίες (random) τιμές για μεταβλητές καταστάσεων. Συνδυάζεται ιδεατά με το "where" που επεξηγήθηκε πιο πάνω. Παράδειγμα χρήσης:

randomgrade:Nat:= choose n (where n>=0 \/ n<101) ;

Αν χρησιμοποιηθεί το choose χωρίς περιορισμό, θα μας δώσει μια τυχαία τιμή από το πεδίο τιμών του συνόλου του τύπου της μεταβλητής. Με άλλα λόγια, αν απαλείψουμε το κομμάτι που βρίσκεται σε παρενθέσεις στο παράδειγμα, η μεταβλητή καταστάσεων "randomgrade" θα πάρει μια τυχαία τιμή από το σύνολο των φυσικών αριθμών.

- **Tuple:** Πλειάδα. Κάτι αντίστοιχο με το struct της γλώσσας C. Μας επιτρέπει να δηλώσουμε πιο σύνθετες μεταβλητές που περιέχουν περισσότερα από 1 στοιχεία. Πρέπει να ορίζονται στην κορυφή του αρχείου όπου βρίσκεται ο κώδικας ή σε βοηθητικά αρχεία. Παράδειγμα χρήσης:

student_record: Tuple [Name:String, Surname:String,Age:Int]

Το πιο πάνω μας επιτρέπει να δηλώσουμε μεταβλητές του τύπου student_record και για να αποκτήσουμε πρόσβαση σε συγκεκριμένα πεδία χρησιμοποιούμε τον χειριστή "." . Για παράδειγμα:

```
student_array : Array [Int , student_record];
student_array[1].Name:="John";
student_array[1].Surname:="Doe";
student_array[1].Age:=15;
```

- **Vocabulary:** Λεξιλόγιο. Κάτι αντίστοιχο με τη δημιουργία βιβλιοθηκών στην γλώσσα C. Η δημιουργία ενός λεξιλογίου μας επιτρέπει να ορίσουμε κάποιους δικούς μας τύπους (πιθανόν σε διαφορετικό αρχείο) και εισάγοντας το λεξιλόγιο στον κώδικα μας αυτόματα οι τύποι αυτοί θα υπάρχουν και θα ισχύουν. Ένα λεξιλόγιο μπορεί να συμπεριλαμβάνει και άλλα λεξιλόγια. Στο παράδειγμα που ακολουθεί φαίνεται η δημιουργία 2 λεξιλογίων, όπου το ένα συμπεριλαμβάνει το άλλο:

```
vocabulary RecordsVocab
   types
        student_rec:Tuple [name:String,surname:string,age:Nat],
        teacher_rec:Tuple [name:String, surname:string,age:Nat]
end

vocabulary ClassVocab
   imports RecordsVocab
   types
        class:Tuple[teacher:teacher_rec,students:array[int,student_rec]]
end
```

Όπως φαίνεται και πιο πάνω, δημιουργώ αρχικά ένα λεξιλόγιο με το όνομα "RecordsVocab", μέσα στο οποίο καθορίζω 2 νέους τύπους δεδομένων, έναν για αρχεία φοιτητή και έναν για αρχεία καθηγητή. Στη συνέχεια ορίζω ένα δεύτερο λεξιλόγιο με το όνομα "ClassVocab", το οποίο περιλαμβάνει μέσα και το λεξιλόγιο "RecordsVocab". Έτσι, μπορώ να δημιουργήσω τον νέο τύπο δεδομένου για ένα τμήμα, ο οποίος αποτελείται από ένα αρχείο καθηγητή και ένα μονοδιάστατο πίνακα με αρχεία φοιτητών. Με την λέξη imports μπορώ επίσης να ορίσω στην αρχή ενός προγράμματος ότι το πρόγραμμα αυτό συμπεριλαμβάνει ένα λεξιλόγιο ή κάποιο άλλο αρχείο. Σημείωση : αν σε ένα πρόγραμμα συμπεριλάβω το λεξιλόγιο "ClassVocab", δεν χρειάζεται να συμπεριλάβω και το "RecordsVocab", μιας και το ίδιο συμπεριλαμβάνεται ήδη στο

"ClassVocab". Μάλιστα, κάτι τέτοιο θα μας δώσει συντακτικό σφάλμα για διπλό ορισμό τύπου.

- **Union:** Ένωση. Τα στοιχεία του τύπου δεδομένων Union[f1:T1,...,fn:Tn] αντιπροσωπεύουν μια τιμή της οποίας ο τύπος πρέπει να είναι ένα από τα T1 ..Tn. Χρησιμοποιείται όταν δεν γνωρίζουμε εκ των προτέρων τι τύπο υ θα είναι ένα αντικείμενο.
- **Conditional Statements:** Τα κλασσικά if-then-else statements όπως αυτά χρησιμοποιούνται στις περισσότερες γλώσσες. Μια if-then-else δήλωση αποτελείται από την λέξη κλειδί "if", ένα κατηγόρημα boolean, την λέξη "then" και ένα τμήμα κώδικα το οποίο θα εκτελείται όταν το κατηγόρημα boolean είναι αληθές. Στη συνέχει ακολουθούν τα τμήματα elseif και else, για τις περιπτώσεις που το κατηγόρημα boolean δεν είναι αληθές. Τα elseif τμήματα περιλαμβάνουν ένα νέο κατηγόρημα boolean, ενώ τα else απλά εκτελούνται όταν το κατηγόρημα boolean του τμήματος "if-then" είναι μη αληθές. Κάθε τμήμα "if-then-else" πρέπει να τελειώνει με την λέξη-κλειδί "fi". Τα τμήματα "if-then-else" μπορεί να είναι και φωλιασμένα. Παράδειγμα χρήσης:

```
if (x > y \/ x > z ) then
        max:=x;
elseif (y > z) then
        max:=y;
else
        max:=z;
fi;
```

- **For Statements:** Τα κλασσικά for loops, έτσι όπως αυτά χρησιμοποιούνται στις περισσότερες γλώσσες προγραμματισμού. Μας επιτρέπει να χρησιμοποιήσουμε επανάληψη στον κώδικα μας. Δομικά, ένα for block αρχίζει με τη λέξη-κλειδί "for" και συνεχίζεται με ένα κομμάτι συνθήκη και την λέξη κλειδί "do". Ακολούθως, περιλαμβάνεται το μέρος του κώδικα που θα εκτελείται σε κάθε επανάληψη. Κάθε τμήμα for πρέπει να τελειώνει με τη λέξη-κλειδί "od". Ισχύει και πάλι ότι πολλά for loops μπορεί να είναι φωλιασμένα. Παράδειγμα χρήσης:

```
for x:mytype in Myset do ..... od
```

Στο πιο πάνω παράδειγμα δημιουργούμε τοπικά μια προσωρινή μεταβλητή χ του τύπου mytype (μπορεί να είναι και τύπος που δημιουργήθηκε από το χρήστη, είτε ως tuple είτε ως enumeration). Ο κώδικας θα εκτελεστεί μια φορά για κάθε στοιχείο τύπου mytype που υπάρχει μέσα στο σύνολο Myset.

- **Let Functions:** Είναι συναρτήσεις που η γλώσσα TIOA επιτρέπει για την δική μας ευκολία. Δομικά, μια συνάρτηση ορίζεται με την λέξη κλειδί "let", το όνομα της συνάρτησης, τα ονόματα και οι τύποι των παραμέτρων, ο τύπος επιστροφής της συνάρτησης και τέλος ο ορισμός της συνάρτησης. Παράδειγμα χρήσης:

let CorrectGrade (grade):Nat -> Bool = grade>=0 \/ grade <101;

Η συνάρτηση αυτή παίρνει ως παράμετρο ένα φυσικό αριθμό (τον οποίο τοπικά ονομάζει grade) και επιστρέφει true όταν ο αριθμός αυτός είναι μεταξύ του 0 και του 100.

2.4.1.3 Σύνθεση ενός αυτόματου TIOA

Όπως περιγράφτηκε και σε προηγούμενη ενότητα, ένα αυτόματο TIOA αποτελείται από τα εξής τμήματα:

1. Υπογραφή Αυτόματου (Action Signature)
2. Μεταβλητές Καταστάσεως (States)
3. Μεταβάσεις (Transitions)
4. Τροχιές (Trajectories)

Τα τμήματα αυτά θα περιγραφούν στην συνέχεια αναλυτικά.

2.4.1.4 Υπογραφή Αυτόματου

Η υπογραφή για ένα αυτόματο (signature) έχει παρόμοια λειτουργία με τον ορισμό των πρωτότυπων στην γλώσσα C, με τη διαφορά ότι στα αυτόματα TIOA είναι απαραίτητο να υπάρχει πάντα. Κάθε αυτόματο, μετά τον ορισμό του, ξεκινά με την υπογραφή. Η υπογραφή περιλαμβάνει ένα ορισμό για όλες τις ενέργειες που θα περιλαμβάνει το αυτόματο, καθώς και τι τύπου είναι (εσωτερικές, εισόδου ή εξόδου) και τι παραμέτρους παίρνει η κάθε μια. Αρχίζει με τη λέξη-κλειδί "signature". Η κάθε περιγραφόμενη ενέργεια ορίζεται ρητά σε επόμενο στάδιο του αυτόματου. Παράδειγμα υπογραφής αυτόματου:

```
automaton Process (PID: Nat) where PID>0
  signature
      input Receive(message:Nat, sender:Nat) where sender<>PID
      output Send (message:Nat,receiver:Nat) where receiver<>PID
      internal beginProcess()
```

2.4.1.5 Μεταβλητές Καταστάσεων

Μετά την υπογραφή του αυτομάτου ακολουθεί το σύνολο των μεταβλητών καταστάσεων (states) του αυτομάτου. Στην ουσία πρόκειται για τις μεταβλητές του αυτομάτου. Ορίζονται από τη λέξη-κλειδί "states" και στη συνέχεια ορίζεται η κάθε μεταβλητή με τον τύπο της και μια αρχικοποίηση (χρησιμοποιώντας τον τελεστή ανάθεσης ":="). Παράδειγμα δήλωσης μεταβλητών καταστάσεων ενός αυτομάτου:

```
states
      receivednumber:Nat:=0;
      replyqueue:Seq[Nat]:={};
      clock:DiscreteReal:=0;
```

2.4.1.6 Μεταβάσεις / Ενέργειες

Σε αυτό το τμήμα του αυτόματου ορίζονται οι μεταβάσεις / ενέργειες, όπως αυτές προκαθορίστηκαν νωρίτερα στην υπογραφή. Κάθε ενέργεια μπορεί προαιρετικά να μια προϋπόθεση για να εκτελεστεί (αυτό δεν ισχύει για τις ενέργειες εισόδου οι οποίες δεν μπορούν να έχουν προϋποθέσεις, μιας και είναι πάντα ενεργές) και την επίδραση της ενέργειας. Το τμήμα αυτό ξεχωρίζει από την λέξη-κλειδί "Transitions" στην αρχή του. Επιπλέον, σε κάθε μετάβαση μπορούμε να δηλώσουμε τοπικές μεταβλητές με την λέξη-κλειδί "locals" στην αρχή της. Παράδειγμα τμήματος μεταβάσεων / ενεργειών για ένα αυτόματο TIOA (έστω αυτό που δείχνουμε στα προηγούμενα παραδείγματα της ενότητας 2.4.1.4 και 2.4.1.5):

```
Transitions
      input Receive (message, sender)
        eff replyqueue := repleyqueue |- sender;
            receivednumber := message;
      output Send (message, receiver)
```

```
locals
 counter:Nat:=0;
 pre replyqueue ~= {};
 eff replyqueue := eject (replyqueue,sender);
     counter:=counter+1;
```

Το πιο πάνω τμήμα κώδικα υλοποιεί τις 2 ενέργειες Receive (εισόδου) και Send (εξόδου). Παρατηρούμε ότι η send θα εκτελεστεί μόνο εφόσον η ουρά replyqueue δεν είναι κενή, και θα αφαιρέσει τον συγκεκριμένο παραλήπτη (receiver) από τη λίστα αυτή. Σημειώνουμε πως σε αυτή την περίπτωση η τοπική μεταβλητή counter θα αρχικοποιείται κάθε φορά με 0, και άρα δεν θα ξεπεράσει ποτέ τον αριθμό 1. Αντίστοιχα, η receive λαμβάνει ένα φυσικό αριθμό (το message), τον οποίο αποθηκεύει στην μεταβλητή καταστάσεως receivednumber. Επίσης, προσθέτει τον αποστολέα του μηνύματος στην ουρά replyqueue για να μπορεί να του απαντήσει αργότερα (δηλαδή για κάθε receive θα μπορεί να εκτελεστεί και ένα send).

2.4.1.7 Τροχιές

Οι τροχιές, όπως αναφέρεται και νωρίτερα, είναι αυτές που υλοποιούν την ιδέα του χρόνου στα αυτόματα της γλώσσας TIOA. Ένα αυτόματο μπορεί να έχει πολλές τροχιές, οι οποίες να κινούνται σε διαφορετικούς ρυθμούς (ωστόσο, είναι λάθος η ίδια τροχιά για 2 διαφορετικά αυτόματα να μην κινείται με τον ίδιο ρυθμό). Το τμήμα των τροχιών αρχίζει πάντα με τη λέξη-κλειδί "Trajectories". Ακολούθως, η κάθε τροχιά ορίζεται με τον εξής τρόπο:

```
trajdef "name"
 (invariant "conditions";)
 (stop when "conditions";)
evolve d("variable")=1;
```

Οι γραμμές κώδικα που βρίσκονται σε παρενθέσεις είναι προαιρετικές. Η σταθερή σχέση (invariant) ισχύει μόνο για την συγκεκριμένη τροχιά και πρέπει να είναι συνέχεια αληθές. Η συνθήκη stop when, μόλις ικανοποιηθεί, τερματίζει αμέσως την συγκεκριμένη τροχιά. Η γραμμή "evolve" μας δίνει το ρυθμό με τον οποίο εξελίσσεται η συγκεκριμένη τροχιά. Για παράδειγμα, αν στο αυτόματο έχει δηλωθεί μια μεταβλητή καταστάσεων με το όνομα "clock" τύπου DiscreteReal για να χρησιμοποιείται ως το φυσικό ρολόι του αυτόματου, μια δήλωση για να εξελίσσεται η τροχιά με ρυθμό 1 θα είναι η εξής: evolve d(clock)=1;

2.4.1.8 Σταθερές Σχέσεις

Αν και προαιρετικά, είναι δυνατό να ορίσουμε κάποιες προϋποθέσεις οι οποίες πρέπει να είναι αληθείς κατά την καθολική διάρκεια εκτέλεσης ενός αυτόματου. Οι σταθερές σχέσεις αυτές (invariants) δηλώνονται ως εξής:

```
invariant "name" of "Automaton_name"
        "condition 1"
            ...
        "condition n"
```

Σύμφωνα με τον ορισμό του, καθ' όλη τη διάρκεια εκτέλεσης του αυτόματου με το όνομα Automaton_name θα πρέπει να ισχύουν και οι n συνθήκες. Οι σταθερές σχέσεις τοποθετούνται συνήθως κάτω από το τμήμα των τροχιών.

2.4.1.9 Σύνθεση Αυτομάτων TIOA

Είναι δυνατό για ένα αυτόματο να περιλαμβάνει πολλά αυτόματα ως μέλη του. Η διαδικασία αυτή ονομάζεται σύνθεση αυτομάτων και μας επιτρέπει να έχουμε επικοινωνία και να συνθέτουμε πιο πολύπλοκα προγράμματα αφού πρώτα τα απλοποιήσουμε και τα δημιουργήσουμε σε πιο ψηλό επίπεδο αφαιρετικότητας. Για παράδειγμα, μπορούμε να δημιουργήσουμε ένα σύστημα επικοινωνίας n κόμβων, δημιουργώντας ένα αυτόματο με τον κώδικα που θα εκτελεί ο κάθε κόμβος (έστω ότι ονομάζουμε το αυτόματο αυτό Node(NodeID)) και ένα άλλο αυτόματο που θα λειτουργεί ως κανάλι επικοινωνίας (έστω ότι ονομάζουμε το αυτόματο αυτό Channel(from,to)). Τώρα, μπορούμε να συνθέσουμε το σύστημα μας δημιουργώντας ένα νέο αυτόματο που θα περιέχει n αυτόματα τύπου Node και 2n αυτόματα τύπου Channel (μιας και τα κανάλια δεν είναι αμφίδρομα θα χρειάζονται 2 για κάθε ζεύγος κόμβων). Το νέο αυτόματο δηλώνεται ως εξής:

```
automaton Composition
  components N1: Node(1), N2:Node(2).....Nn:Node(n);
                C12:Channel(1,2), C21:Channel(2,1)....Cn.n-1:Channel(n,n-1);
```

2.4.2 Προγραμματισμός των μεταβάσεων και των τροχιών

Αφού δημιουργήσουμε μια σύνθεση, φτάνουμε στο σημείο όπου θέλουμε να γράψουμε ένα πρόγραμμα εκτέλεσης που να χρησιμοποιεί τις προδιαγραφές μας σε κάποιο σενάριο. Το κομμάτι αυτό του κώδικα αρχίζει με την λέξη-κλειδί "schedule". Η σύνταξη για το μέρος αυτό του κώδικα είναι παρόμοια με αυτή που γράφονται τα υπόλοιπα τμήματα ενός αυτομάτου TIOA, με τη χρήση κάποιων επιπρόσθετων εντολών. Συγκεκριμένα, οι εντολές αυτές είναι οι "follow" για να εξελίξουμε κάποιες τροχιές, η "print" για να εκτυπώσουμε την τιμή κάποιας μεταβλητής καταστάσεων (για σκοπούς αποσφαλμάτωσης ή επιβεβαίωσης ορθότητας του κώδικα) και η "fire" για να πυροδοτήσουμε κάποια μεταβατική ενέργεια. Παράδειγμα σωστής χρήσης:

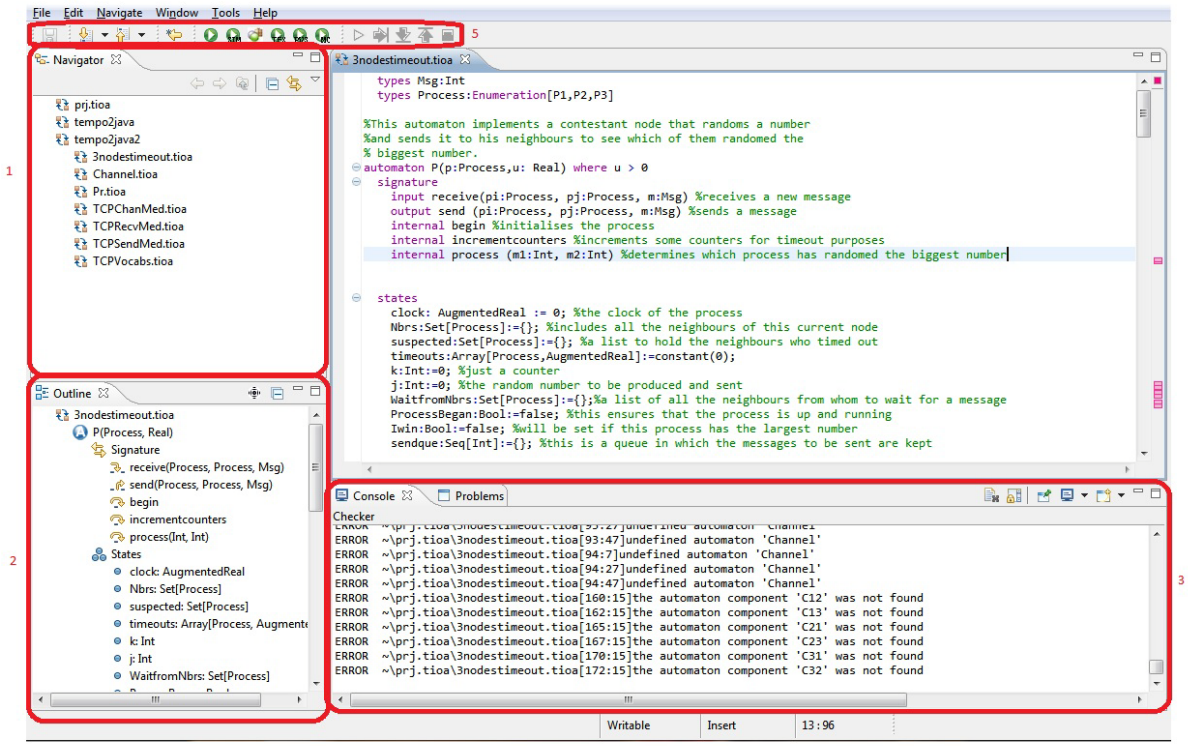
```
follow N1.traj, N2.traj, ... , Nn.traj duration 1;
fire input N1.receive(msg,2);
print (N1.tempvariable)
```

Για το υπόλοιπο μέρος του "schedule" μπορεί να οριστούν μεταβλητές καταστάσεων, και η υλοποίηση του σεναρίου προσομοίωσης μπορεί να πραγματωθεί με χρήση συνθηκών και βρόγχων. Ένα ολοκληρωμένο πρόγραμμα μεταβάσεων υπάρχει στην επόμενη ενότητα, όπου δείχνω ένα ολοκληρωμένο αλγόριθμο υλοποιημένο στο εργαλείο TEMPO.

2.5 Παράδειγμα Υλοποίησης ενός Απλού Αλγορίθμου στο Εργαλείο TEMPO

2.5.1 Αρχίζοντας με το εργαλείο TEMPO

Το εργαλείο TEMPO μπορεί ο καθένας να το κατεβάσει και να το χρησιμοποιήσει από την επίσημη ιστοσελίδα του [7]. Στην ιστοσελίδα υπάρχουν οδηγίες για τη σωστή εγκατάσταση και χρήση του εργαλείου (Σημ.: Πιθανό να χρειαστεί και εγκατάσταση του Java SDK ή JRE αν δεν υπάρχει ήδη, αυτό γίνεται από την επίσημη ιστοσελίδα http://www.java.sun.com). Αφού γίνει σωστά η εγκατάσταση, το εργαλείο θα φαίνεται όπως στο σχήμα 2.5.1.1.



Σχήμα 2.5.1.1 Το εργαλείο TEMPO

Τα μέρη της διαφάνειας χρήστη στο εργαλείο TEMPO, όπως αυτά φαίνονται και αριθμούνται στο σχήμα 2.5.1.1, περιγράφονται περιληπτικά πιο κάτω:

1. **Navigator:** Εδώ φαίνονται όλες οι εργασίες (projects) που βρίσκονται στον χώρο εργασίας μας (workspace). Μπορούμε από εδώ να ανοίξουμε διάφορα αρχεία. Τα αρχεία ανοίγουν στον editor(3).
2. **Outline:** Εδώ φαίνεται ένα επίγραμμα του τρέχοντος αρχείου που έχουμε ανοικτό. Φαίνονται τα τμήματα του αυτόματου και οι τιμές για τις μεταβλητές καταστάσεων. Είναι χρήσιμο στις προσομοιώσεις για επαλήθευση και επίσης μας βοηθά να "δούμε" τον σχεδιασμό μας οπτικά και να εντοπίσουμε τυχόν ατέλειες στον σχεδιασμό.
3. **Console:** Η κονσόλα του συστήματος. Αν και ονομάζεται κονσόλα, δεν δέχεται εντολές, μόνο τυπώνει. Εδώ φαίνονται τυχόν σφάλματα στην μετάφραση (στο σχήμα αφαίρεσα επίτηδες τον ορισμό του αυτόματου "Channel" για να φανούν μερικά σφάλματα) ή τα αποτελέσματα της τρέχον προσομοίωσης. Επίσης, μετά από επιτυχημένη μετάφραση του κώδικα σε java με τη χρήση του εργαλείου tempo2java στην κονσόλα θα φανεί ο πηγαίος κώδικας java που παράχθηκε.
4. **Editor:** Από εδώ μπορούμε να γράψουμε / μεταβάλουμε τον κώδικα που περιλαμβάνεται σε ένα αρχείο. Πολλά αρχεία μπορούν να είναι ανοικτά ταυτόχρονα.
5. **Control Bar:** Περιλαμβάνει διάφορα κομβία που μας βοηθούν στη λειτουργία του TEMPO. Τα πιο βασικά φαίνονται πιο κάτω:

- Ελεγκτής (Checker ) : Ενεργοποιεί τον ελεγκτή, ο οποίος ελέγχει το τρέχον αρχείο για συντακτικά ή σημαντολογικά σφάλματα.
- Προσομοιωτής (Simulator ) : Τρέχει μια προσομοίωση της εργασίας χρησιμοποιώντας τον προσομοιωτή του εργαλείου TEMPO.
- Μεταφραστής Java (tempo2java plugin ) : Μεταφράζει τον κώδικα μας σε Java. Τα αρχεία που παράγονται μπορούν να βρεθούν στον χώρο εργασίας (όπου βρίσκεται και ο ορισμός των αυτομάτων που μεταφράστηκαν) σε ένα νέο φάκελο με το όνομα "Projectname.java" όπου "Projectname" είναι το αντίστοιχο όνομα της εργασίας που μεταφράστηκε.
- Κομβία ρυθμίσεως προσομοίωσης (Simulation Controllers ) : Ενεργοποιούνται μόνο κατά τη διάρκεια μιας προσομοίωσης. Μας επιτρέπουν να ελέγξουμε τη ροή της προσομοίωσης είτε βήμα-βήμα είτε να προχωρούμε την προσομοίωση σε συγκεκριμένα προκαθορισμένα από εμάς σημεία (break points).

Για να δημιουργήσουμε μια νέα εργασία (στην ουσία ένα "δοχείο" στο οποίο θα περιέχονται όλα τα αρχεία της συγκεκριμένης εργασίας) πάμε από το file menu -> new Project. Στη συνέχεια μπορούμε να δούμε την εργασία αυτή στον navigator. Από εκεί μπορούμε να εισάξουμε (import) ήδη υπάρχοντα αρχεία από άλλες εργασίες ή να δημιουργήσουμε καινούρια. Είναι όμως απαραίτητο για κάθε αρχείο που περιγράφει προδιαγραφές ενός αυτομάτου / προσομοίωσης σε γλώσσα TIOA να έχει και την επέκταση ".tioa".

2.5.2 Παράδειγμα υλοποίησης ενός απλού αλγορίθμου

Περιγραφικά, ο αλγόριθμος που θα δείξω λέει τα εξής: "Έστω ότι έχω 3 κόμβους. Οι 3 κόμβοι αυτοί σε κάθε επανάληψη δημιουργούν από ένα τυχαίο αριθμό. Ο κάθε κόμβος στη συνέχεια στέλνει τον αριθμό του στους άλλους 2 κόμβους. Πριν το τέλος κάθε επανάληψης, ο κάθε κόμβος θα πρέπει να αναγνωρίσει αν έχει δημιουργήσει το μεγαλύτερο από τους 3 αριθμούς που δημιουργήθηκαν."

Για να υλοποιήσουμε αυτό τον αλγόριθμο αφαιρετικά, θα χρειαστούμε ένα μοντέλο αυτομάτου TIOA το οποίο θα υλοποιεί ένα κόμβο, και ένα μοντέλο αυτομάτου που θα υλοποιεί το κανάλι. Μιας και στην υλοποίηση της διπλωματικής αυτής εργασίας θα χρησιμοποιηθεί το Message Passing Interface (MPI), θα χρησιμοποιήσουμε τα έτοιμα αρχεία καναλιών MPI που παρέχονται για την επικοινωνία μεταξύ των κόμβων . Για το λόγο αυτό

δεν θα δείξω τους κώδικες για τα κανάλια, οι οποίοι είναι ήδη υλοποιημένοι. Ωστόσο, οι κώδικες αυτοί μπορούν να βρεθούν στο Παράρτημα Α. Χρειάζεται μόνο να δείξω τον εσωτερικό κώδικα ενός κόμβου, τον οποίο θα αποκαλέσω "process" καθώς και τον ολοκληρωμένο κόμβο (ο εσωτερικός κώδικας μαζί με κάποιο μέσο επικοινωνίας με άλλους κόμβους), τον οποίο θα αποκαλέσω "composition" για το λόγο ότι στην ουσία αποτελεί σύνθεση άλλων αυτομάτων.

```
automaton Process(u: DiscreteReal, myrank: Nat)
  signature
    output SEND( m:Null[MPI_message] )
    input  RECEIVE( m:Null[MPI_message] )
    internal prepmessages, init ,process

  states
    mynumber: DiscreteReal:=0; %the random number to be produced and sent
    Iwin:Bool:=false; %will be set if this process has the largest number
    maxother:  DiscreteReal := 0; %the biggest of the numbers received
    physclock: DiscreteReal := 0; %the physical clock of the automaton
    tosend:    Seq[Null[MPI_message]] := { }; %the send queue
    Nbrs:       Seq[Nat] := { }; %includes all the neighbors of this process
    initially u >= 0;
```

Σχήμα 2.5.2.1 Προδιαγραφή του εσωτερικού κόμβου

Η υπογραφή και οι μεταβλητές καταστάσεων, καθώς και μια απλή σταθερή σχέση φαίνονται στο σχήμα 2.5.2.1. Πριν προχωρήσω σε περαιτέρω ανάλυση αξίζει να αναφέρω ότι για την υλοποίηση του MPI καναλιού χρησιμοποίησα τον έτοιμο κώδικα που ανάρτησα στο Παράρτημα Α. Ο τύπος δεδομένων MPI_message ορίζεται στο λεξιλόγιο του MPIVocabs.tioa αρχείο, το οποίο φαίνεται στο Παράρτημα Α.3. Χρησιμοποιείται ως ο τύπος του μηνύματος που θα αποσταλεί. Αποτελείται από μια πλειάδα, η οποία συντίθεται από ένα τμήμα message (DiscreteReal) και 2 μεταβλητές τύπου Nat, τις sender και receiver για τον αποστολέα και παραλήπτη αντίστοιχα. Να σημειώσουμε εδώ πως οι διεργασίες που θα δημιουργηθούν θα έχουν ένα αύξων αριθμό, κάτι αντίστοιχο με την ταυτότητα διεργασιών (process ID) το οποίο αποκαλείται "rank" και επιστρέφεται από την κλήση της συνάρτησης MPI_Rank(). Επίσης, η συνάρτηση MPI_Size() μας επιστρέφει το σύνολο των διεργασιών που θα δημιουργηθούν.

```
transitions
    output SEND(m)
    pre tosend ~= { };
       m = head(tosend);
    eff tosend := tail(tosend);

  input RECEIVE(m)
    eff
      if m ~= nil() then
         maxother := max(maxother, val(m).data);
      fi

  internal prepmessages
    pre len(tosend) = 0;
    eff
     Iwin:=false; maxother :=0; mynumber:= choose n where n>0;
      for n : Nat where n < len(Nbrs) do
         tosend := tosend |- embed([mynumber,myip,Nbrs[n]]);
      od



  internal init
  locals
     temp:Nat:=0;
     pre true;
    eff
       for n : Nat where n < MPI_Size() do
           if ~(n=MPI_Rank()) then Nbrs:= Nbrs |- n; fi;
       od
    internal process()
      eff
           if(mynumber > maxnumber) then
                  Iwin:=true;
           fi;


trajectories
  trajdef T
    evolve d(physclock) = u;
```

Σχήμα 2.5.2.2 Υλοποίηση του Εσωτερικού αυτόματου

- **u:** Μια από τις παραμέτρους του αυτόματου. Μας δίνει τον ρυθμό εξέλιξης της τροχιάς (για παράδειγμα, όταν το u=1 , κάθε επανάληψη του αλγόριθμου θα προχωρά για 1 μονάδα χρόνου).
- **myrank:** Η άλλη παράμετρος του αυτόματου. Είναι στην ουσία ο αύξων αριθμός της διεργασίας (που επιστρέφεται από τη συνάρτηση MPI_rank()).
- **mynumber:** Τύπου DiscreteReal. Χρησιμοποιείται για την δημιουργία και προσωρινή αποθήκευση του τυχαίου αριθμού σε κάθε επανάληψη.
- **Iwin:** Τύπου boolean. Στην αρχή κάθε επανάληψης την θέτουμε σε θέση "false". Γίνεται "true" μόνο όταν το αυτόματο λάβει τους 2 άλλους αριθμούς και καθορίσει ότι ο δικός του αριθμός είναι ο μεγαλύτερος.
- **maxother:** Τύπου DiscreteReal. Χρησιμοποιείται για προσωρινή αποθήκευση του μεγαλύτερου από τους 2 αριθμούς που λαμβάνει το αυτόματο σε κάθε επανάληψη.
- **physclock:** Τύπου DiscreteReal. Αντιπροσωπεύει το φυσικό ρολόι της διεργασίας.
- **tosend:** Ουρά τύπου MPI_message. Εδώ δημιουργούνται και αποθηκεύονται τα προς αποστολή μηνύματα σε κάθε επανάληψη.
- **Nbrs:** Ουρά τύπου Nat. Φυλάει το σύνολο των κόμβων που υπάρχουν στο σύστημα (δηλαδή τον αριθμό MPI_rank() του κάθε κόμβου).

Στο σχήμα 2.5.2.2 φαίνεται η υλοποίηση των μεταβάσεων, όπως αυτές ορίστηκαν στην υπογραφή του αυτομάτου (σχήμα 2.5.2.1). Περιληπτικά, η κάθε ενέργεια περιγράφεται πιο κάτω:

- **output SEND(m:Null[MPI_message]):** Η μετάβαση αυτή στέλνει το μήνυμα που βρίσκεται στην κορυφή της ουράς "tosend". Προϋποθέτει ότι η ουρά δεν είναι κενή. Επειδή είναι ενέργεια εξόδου, με την πυροδότηση της ενεργοποιεί την ενέργεια εισόδου send για άλλα αυτόματα (στο παράδειγμα αυτό θα ενεργοποιήσει την ενέργεια εισόδου "send" για το κανάλι MPI).
- **input RECEIVE(m:Null[MPI_message]):** Η μετάβαση αυτή λαμβάνει ένα μήνυμα από το κανάλι. Επίσης, συγκρίνει την τιμή που λαμβάνει με αυτή της μεταβλητής καταστάσεων "maxother", και αν η τιμή που λαμβάνει είναι μεγαλύτερη την αντικαθιστά. Στην ουσία, όταν το αυτόματο λάβει τα 2 μηνύματα σε κάθε επανάληψη, η μεταβλητή καταστάσεων "maxother" θα περιέχει τον μεγαλύτερο από τους 2 αριθμούς που έστειλαν οι άλλες 2 διεργασίες.
- **internal prepmessages():** Η μετάβαση αυτή αποτελεί το κύριο μέρος της εσωτερικής λειτουργίας του αυτόματου. Αρχικά, αρχικοποιεί τις μεταβλητές καταστάσεων "Iwin" σε "false" (σε περίπτωση που στην προηγούμενη επανάληψη η διεργασία είχε τον

μεγαλύτερο αριθμό) και "maxother" σε 0 (για να μπορεί να λάβει τους 2 νέους αριθμούς της νέας επανάληψης). Στην συνέχεια παράγει τον νέο τυχαίο αριθμό χρησιμοποιώντας μια εντολή "choose". Τέλος, δημιουργεί ένα μήνυμα προς αποστολή για κάθε μια από τις υπόλοιπες διεργασίες που βρίσκονται στην λίστα "tosend", επιτρέποντας έτσι την κλήση της ενέργειας εξόδου που θα στείλει τα μηνύματα.

- **internal init:** Αρχικοποιεί την ουρά δική της ουρά "Nbrs" εισάγοντας τις τιμές των ταυτοτήτων (rank) των υπόλοιπων διεργασιών. Αυτό το κάνει με χρήση των συναρτήσεων MPI_Rank() και MPI_Size() που επιστρέφουν την ταυτότητα της συγκεκριμένης διεργασίας και το πλήθος των διεργασιών αντίστοιχα. Χρειάζεται για να μπορούμε αργότερα να θέσουμε κάποιες από τις διεργασίες ως "μη ενεργές".
- **internal process (maxnumber:DiscreteReal):** Η μετάβαση αυτή απλά θέτει την μεταβλητή καταστάσεων "Iwin" σε "true" κατάσταση όταν ο αριθμός που δημιούργησε η διεργασία αυτή είναι μεγαλύτερος από τον μεγαλύτερο από τους 2 αριθμούς που παρέλαβε στην συγκεκριμένη επανάληψη.

Τέλος, στο σχήμα 2.5.2.2 φαίνεται ο ορισμός της τροχιάς του εσωτερικού αυτόματου. Η τροχιά αυτή είναι μια απλή, βασική τροχιά η οποία προχωρεί με ρυθμό u (u μονάδες χρόνου ανά επανάληψη - το u δίνεται σαν παράμετρος στο αυτόματο).

Σε αυτό το σημείο έχει τελειώσει ο ορισμός του εσωτερικού αυτόματου. Το αυτόματο μπορεί τώρα να αποθηκευτεί, έστω ότι ονομάζουμε το αρχείο αυτό "process.tioa". Χρειάζεται όμως να οριστεί και το αυτόματο σύνθεσης, το οποίο θα περιλαμβάνει ένα εσωτερικό αυτόματο "process" και κάποια άλλα αυτόματα που θα υλοποιήσουν το μέσο επικοινωνίας του αυτόματου μέσω του πρωτοκόλλου MPI. Το πρώτο βήμα για αυτό το νέο αυτόματο θα είναι να συμπεριλάβουμε τα απαραίτητα αρχεία. Αυτό φαίνεται στο σχήμα 2.5.2.3.

```
%%% .: VOCABS :.
include "MPIVocabs.tioa"

%%% .:TCP mediator automata:.
include "ReceiveMediator.tioa"
include "SendMediator.tioa"


%%% .: Process Automaton :.
include "Process.tioa"
```

Σχήμα 2.5.2.3 Εισαγωγή των απαραίτητων αρχείων

Αφού εισάγουμε τα απαραίτητα αρχεία, μπορούμε τώρα να ορίσουμε το σύνθετο αυτόματο. Ο ορισμός του φαίνεται στο σχήμα 2.5 2.4 . Το σύνθετο αυτόματο αποτελείται από τα αυτόματα SM και RM τα οποία αποτελούν το κανάλι MPI, καθώς και ένα αυτόματο PR τύπου "Process", όπως αυτό ορίστηκε πιο πάνω. Παρατηρούμε επίσης ότι το αυτόματο παίρνει ως παράμετρο το u (ρυθμός αύξησης της τροχιάς). Την παράμετρο αυτή την περνά, καθώς και το "rank", στο αυτόματο PR.

```
automaton Composition(u: DiscreteReal)
       components
    PR : Process(u,MPI_Rank);
       SM : SendMediator;
       RM : ReceiveMediator;
```

Σχήμα 2.5.2.4 Ορισμός του σύνθετου αυτόματου

Το σύνθετο αυτόματο δεν χρειάζεται δικές του μεταβάσεις. Όλες οι απαραίτητες ενέργειες υλοποιήθηκαν στα αυτόματα-μέλη του. Το μόνο που χρειαζόμαστε είναι ένα πρόγραμμα μεταβάσεων και τροχιών (schedule). Στο πρόγραμμα αυτό θα πρέπει να ορίσουμε ρητά πόσες επαναλήψεις του αλγορίθμου θέλουμε να εκτελεστούν (στο σχήμα 2.5.2.5 ορίζω 100 επαναλήψεις) και πόσοι είναι οι συμμετέχοντες (αυτό ορίζεται όταν μεταφραστεί το πρόγραμμα με χρήση της βιβλιοθήκης MPJ Express - περισσότερα στο κεφάλαιο 4.1).

```
schedule
states
   m         :Null[MPI_message] := nil();
   dowhile   :Bool := true;
   error     :Null[JVMError] := nil();
   IPS       :Seq[Nat] := { };
   runs      :Nat := 100; % number of algorithm iterations

fire internal PR.init;
```

Σχήμα 2.5.2.5 Ορισμός και μεταβλητές καταστάσεων για την προσομοίωση

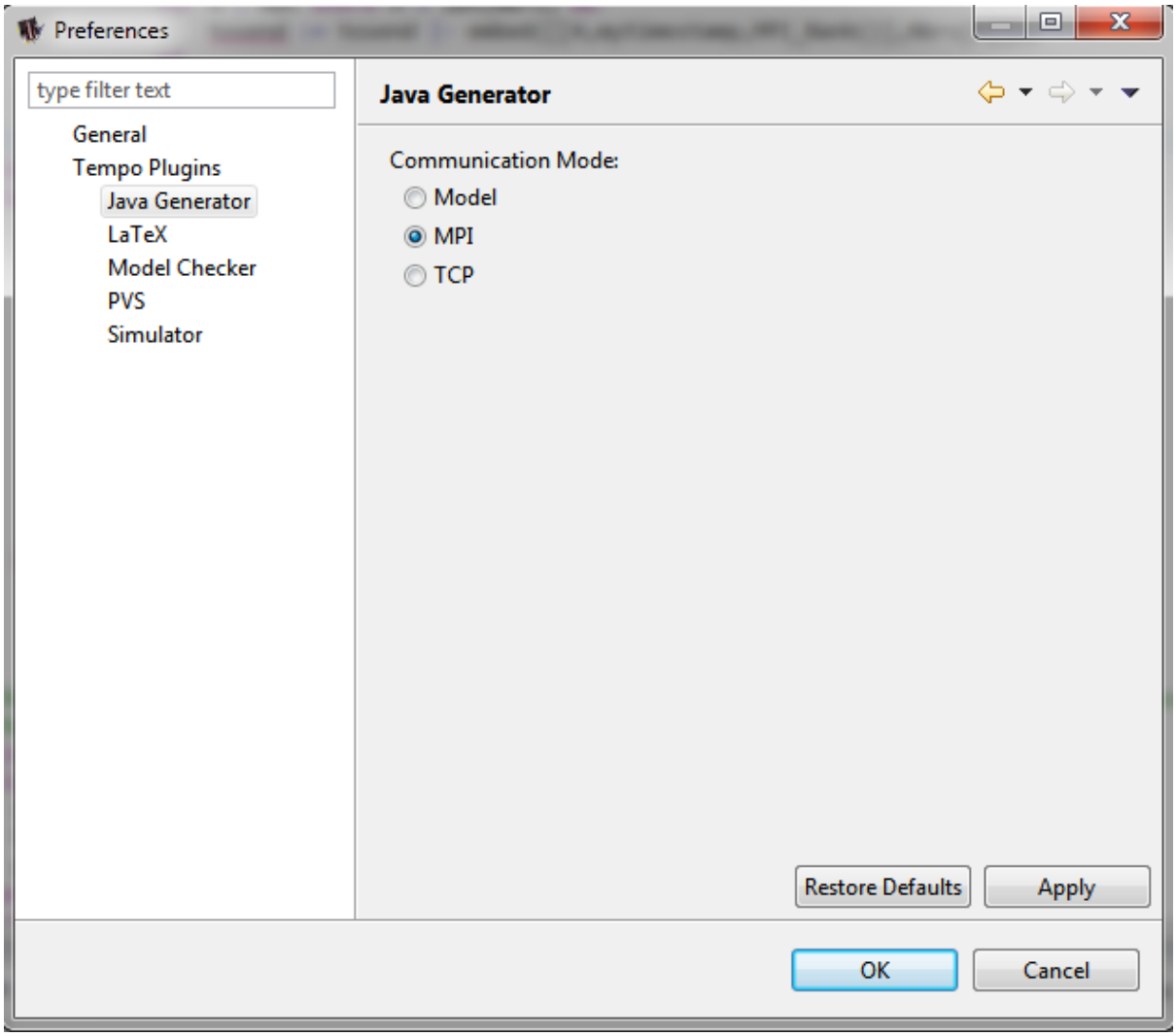
Στο σχήμα 5.2.5.6 φαίνεται το κυρίως μέρος της επανάληψης. Πρόκειται για ένα for loop το οποίο θα εκτελεστεί "runs" φορές. Σε κάθε επανάληψη εκτελείται η εσωτερική ενέργεια "preparemessages" του εσωτερικού κόμβου "Process", η οποία προετοιμάζει τα μηνύματα και αρχικοποιεί τις μεταβλητές κατάλληλα. Ακολούθως, στέλνονται τα μηνύματα και μετά η διεργασία λαμβάνει τυχόν μηνύματα που υπάρχουν στο κανάλι. Τέλος, εκτελείται η εσωτερική ενέργεια "process", η οποία ελέγχει αν ο αριθμός της διεργασίας αυτής είναι ο μεγαλύτερος. Η τιμή της μεταβλητής boolean "Iwin" τυπώνεται στην οθόνη.

```
for i : Nat where i < runs do
      follow PR.T duration u;

      fire internal PR.prepmessages;

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
         od

         % -- process the messages
         fire internal PR.process(PR.maxother);
         print (PR.Iwin);
   od % for on execution length

od % end of schedule
```

Σχήμα 5.2.5.6 Το κυρίως μέρος του Schedule

2.5.3 Μετάφραση του κώδικα σε Java

Αφού τελειώσουμε την προδιαγραφή του αλγορίθμου, φτάνουμε στο σημείο όπου πρέπει να μεταφράσουμε τον κώδικα μας σε Java για να μπορέσουμε να τον τρέξουμε [5]. Μιας και ο πιο πάνω αλγόριθμος χρησιμοποιεί το πρωτόκολλο MPI, θα πρέπει και εμείς να το καθορίσουμε στο tempo2java applet. Αυτό γίνεται πηγαίνοντας στο μενού Window -> Preferences. Από εκεί πηγαίνουμε στο tempo plugins -> java generator και επιλέγουμε την επιλογή MPI, όπως φαίνεται και στο σχήμα 2.5.3.1. Στη συνέχεια πατούμε OK. Τέλος, πατούμε το κουμπί tempo2java που βρίσκεται στη θήκη εργαλείων. Ο κώδικας τώρα έχει παραχθεί και βρίσκεται στον χώρο εργασίας, σε ένα νέο φάκελο με όνομα "3nodescontest.java". Τώρα, μπορούμε να εισάξουμε τον κώδικα αυτό σε κάποιο περιβάλλον java, όπως είναι το Eclipse toolkit και να το μεταφράσουμε / εκτελέσουμε σαν κώδικα java.



Σχήμα 2.5.3.1 Ρυθμίσεις tempo2java

Κεφάλαιο 3

Προσδιορισμός των αλγορίθμων που αναπτύχθηκαν

3.1 Ο Αμοιβαίος Αποκλεισμός σε Κατανεμημένα Συστήματα

Το πρόβλημα του αμοιβαίου αποκλεισμού αναφέρεται στο πρόβλημα της επιστήμης Πληροφορικής όπου 2 διεργασίες / νήματα / κόμβοι σε ένα κοινό περιβάλλον αποκτούν ταυτόχρονη πρόσβαση στην "κρίσιμη περιοχή" τους (αυτή μπορεί να είναι κοινή μνήμη ή κάποιος άλλος κοινός πόρος) [2]. Δημιουργείται έτσι πιθανότητα σφάλματος στα δεδομένα μας, αφού 2 διαφορετικές διεργασίες είναι πιθανό να προσπαθήσουν για παράδειγμα η μια να γράψει σε μια θέση μνήμης την ίδια ώρα που η άλλη προσπαθεί να διαβάσει την προηγούμενη τιμή. Το πρόβλημα αυτό αναγνωρίστηκε για πρώτη φορά το 1965 από τον διάσημο Ολλανδό Edsger W. Dijkstra. Για τη λύση του προβλήματος του αμοιβαίου αποκλεισμού παρουσιάζεται τόσο σε συστήματα ενός επεξεργαστή όσο και σε κατανεμημένα συστήματα. Στην διπλωματική αυτή εργασία θα ασχοληθούμε με τον αμοιβαίο αποκλεισμό σε κατανεμημένα συστήματα. Αξίζει να σημειώσουμε πως σε κατανεμημένα συστήματα δεν υπάρχει κοινό ρολόι, κοινή μνήμη ή κοινός πυρήνας. Επομένως, το πρόβλημα προκύπτει στη χρήση κοινών πόρων και δεν μπορεί να λυθεί με τη χρήση σημαφόρων, όπως γίνεται για παράδειγμα στα λειτουργικά συστήματα (συστήματα ενός επεξεργαστή). Για τη λύση του προβλήματος αυτού προτάθηκαν διάφοροι αλγόριθμοι. Στην επόμενη ενότητα, και κατ' επέκταση στη διπλωματική αυτή εργασία, θα ασχοληθούμε με δυο συγκεκριμένους αλγόριθμους, τον αλγόριθμο του Lelann για τοπολογίες δακτυλίου και τον αλγόριθμο του Lamport για πλήρως συνδεδεμένες τοπολογίες.

Οι 2 προαναφερόμενοι αλγόριθμοι κατατάσσονται στην κατηγορία των αλγορίθμων που προσεγγίζουν τον αμοιβαίο αποκλεισμό με κατανεμημένη προσέγγιση και όχι με

συγκεντρωτική προσέγγιση. Στην συγκεντρωτική προσέγγιση ένας κόμβος διαφέρει από τους άλλους (ορίζεται ως αρχηγός) και στην ουσία συντονίζει τις υπόλοιπες λειτουργίες ούτως ώστε να αποφύγουν τον αμοιβαίο αποκλεισμό. Ως φυσικό επακόλουθο, ο κόμβος - αρχηγός υποφέρει από συμφόρηση, αφού όλοι οι κόμβοι θέλουν να επικοινωνήσουν μαζί του για να πάρουν άδεια να μπουν στην κρίσιμη περιοχή. Όσο πιο μεγάλο γίνεται το δίκτυο, τόσο λιγότερο αποδοτική είναι η προσέγγιση αυτή. Αντίθετα, στην κατανεμημένη προσέγγιση, η απόφαση για το ποιος κόμβος θα μπει κάθε φορά στην κρίσιμη περιοχή παίρνεται από όλους τους κόμβους συγκεντρωτικά. Ως συνεπακόλουθο, όλοι οι κόμβοι εκτελούν στην ουσία τον ίδιο κώδικα, και ο υπολογισμός γίνεται πλήρως παράλληλα. Για να πάρουν μια απόφαση, οι κόμβοι ανταλλάσσουν μηνύματα και χρησιμοποιούν χρονοσφραγίδες για να ορίσουν ποιος κόμβος κάθε φορά δικαιούται να μπει στην κρίσιμη περιοχή.

Για να θεωρηθεί ένας αλγόριθμος αμοιβαίου αποκλεισμού ορθός πρέπει να καλύπτει κάποιες βασικές προϋποθέσεις, τις οποίες αναλύω επιγραμματικά παρακάτω:

- **Ασφάλεια (safety):** Ο αλγόριθμος πρέπει να μας εγγυάται ότι δεν μπορούν ποτέ 2 κόμβοι να έχουν ταυτόχρονη πρόσβαση στην κρίσιμη περιοχή τους.
- **Βιωσιμότητα (liveliness):** Ο αλγόριθμος πρέπει να μας εγγυάται ότι όλες οι αιτήσεις θα ικανοποιηθούν σε πεπερασμένο αριθμό βημάτων. Με άλλα λόγια, δεν πρέπει να έχουμε παρατεταμένη στέρηση (όταν ένας κόμβος αιτείται να μπει στην κρίσιμη περιοχή αλλά για κάποιο λόγο συνεχώς προηγούνται άλλοι κόμβοι, αφήνοντας τον να περιμένει επ' άπειρον) ούτε αδιέξοδα (deadlocks).
- **Διάταξη (ordering):** Οι κόμβοι οι οποίοι αιτούνται πρόσβαση στην κρίσιμη περιοχή πρέπει να εξυπηρετούνται με κάποια σειρά.

Τέλος, οι κατανεμημένοι αλγόριθμοι αξιολογούνται με κριτήρια επίδοσης τον αριθμό των μηνυμάτων που χρειάζονται καθώς και για την καθυστέρηση που πρέπει να περιμένει ο κάθε κόμβος μέχρι να μπει στην κρίσιμη περιοχή.

3.2 Προσέγγιση με Χρήση Σκυτάλης - Ο Αλγόριθμος του Lelann

Ο αλγόριθμος αυτός χρησιμοποιεί ένα ειδικό μήνυμα, την "σκυτάλη" ή "token", του οποίου μόνο ο κάτοχος δικαιούται πρόσβαση στην κρίσιμη περιοχή [2]. Για να λειτουργήσει σωστά ο αλγόριθμος αυτός απαιτεί το δίκτυο να είναι λογικά οργανωμένο σε δακτύλιο (η φυσική οργάνωση μπορεί ωστόσο να είναι διαφορετική). Στην λογική οργάνωση, ο κάθε κόμβος

πρέπει να γνωρίζει τον επόμενο το ῳ και η σκυτάλη κυκλοφορεί πάντα προς την ίδια κατεύθυνση.

Όταν μια διεργασία p_i λάβει τη σκυτάλη
    Αν δεν θέλει να μπει στην κρίσιμη περιοχή
        Προωθεί αμέσως την σκυτάλη στην $p_{(i+1) \bmod n}$
    Αν θέλει να μπει στην κρίσιμη περιοχή
        Κρατάει τη σκυτάλη
        Μπαίνει στην κρίσιμη περιοχή (1 μόνο φορά)
        Μόλις τελειώσει την χρήση της κρίσιμης περιοχής
            Μεταβιβάζει τη σκυτάλη στην $p_{(i+1) \bmod n}$
Τέλος αλγόριθμου

Σχήμα 3.2.1 Ο αλγόριθμος του Lelann

Ο αλγόριθμος του Lelann φαίνεται στο σχήμα 3.2.1. Κάθε φορά που μια διεργασία επιθυμεί να εισέλθει στην κρίσιμη περιοχή πρέπει να περιμένει τη σκυτάλη. Όταν την λάβει, εισέρχεται στην κρίσιμη περιοχή, εκτελεί τις λειτουργίες που πρέπει να εκτελέσει και αφού βγει από την κρίσιμη περιοχή προωθεί στην σκυτάλη. Αν η διεργασία που λαμβάνει τη σκυτάλη δεν θέλει να εισέλθει στην κρίσιμη περιοχή τότε προωθεί αμέσως την σκυτάλη στον επόμενο κόμβο του λογικού δακτυλίου.

Ο αλγόριθμος αυτός ικανοποιεί τα 3 κριτήρια που προαναφέρονται για τη σωστή υλοποίηση μιας λύσης για το πρόβλημα του αμοιβαίου αποκλεισμού. Το κριτήριο της ασφάλειας ικανοποιείται από το γεγονός ότι υπάρχει μόνο μια σκυτάλη, και άρα μόνο μια διεργασία μπορεί ανά πάσα στιγμή να βρίσκεται εντός της κρίσιμης περιοχής. Τα κριτήρια της βιωσιμότητας και της διάταξης επίσης τηρούνται: το κριτήριο της διάταξης είναι φανερά σε ισχύ, μιας και οι διεργασίες βρίσκονται λογικά παραταγμένες σε δακτύλιο. Όσο για το κριτήριο της βιωσιμότητας, μιας και κάθε κόμβος που επιθυμεί να μπει στην κρίσιμη περιοχή πρέπει να περιμένει το πολύ για n-1 μηνύματα μέχρι να πάρει τη σκυτάλη, δεν υπάρχει θέμα παρατεταμένης στέρησης. Σε αυτό συμβάλλει επίσης το γεγονός ότι κανένας κόμβος δεν μπορεί να μπει στην κρίσιμη περιοχή με την σκυτάλη δυο φορές, αλλά πρέπει να προωθήσει την σκυτάλη μετά από κάθε είσοδο του στην κρίσιμη περιοχή.

Ωστόσο, ο αλγόριθμος αυτός έχει δυο ελαττώματα. Υπάρχουν δυο περιπτώσεις στις οποίες ο αλγόριθμος αυτός μπορεί να παρουσιάσει σφάλμα:

1. Αποτυχία τυχαίας διεργασίας: Σε περιπτώσεις δικτύων, υπάρχει πάντα η πιθανότητα μια διεργασία απλά να σταματήσει να δουλεύει, είτε λόγω μηχανικής βλάβης είτε βλάβης του δικτύου. Στην περίπτωση αυτή θα πρέπει να αναδιοργανωθεί ο λογικός δακτύλιος που χρειάζεται ο αλγόριθμος. Η λύση είναι να θέσουμε ως κανόνα το εξής: κάθε φορά που μια διεργασία $p_{(i+1) \bmod n}$ λαμβάνει τη σκυτάλη θα πρέπει να στέλνει ένα μήνυμα επιβεβαίωσης <acknowledgement> στην διεργασία p_i που της έστειλε την σκυτάλη. Η διεργασία p_i αντίστοιχα θα πρέπει να αναμένει αυτό το <acknowledgement> μήνυμα, και αν δεν το λάβει εντός ενός λογικού χρονικού περιθωρίου πρέπει να υποθέσει ότι η διεργασία αυτή είναι "νεκρή", και να προωθήσει την σκυτάλη στην αμέσως επόμενη "ζωντανή" διεργασία. Δυστυχώς, αυτό προϋποθέτει πως η κάθε διεργασία γνωρίζει ολόκληρη την διοργάνωση του λογικού δακτυλίου. Έτσι, μια "νεκρή" διεργασία που επανέρχεται σε λειτουργία πρέπει απλώς να ενημερώσει την προηγούμενη της.
2. Απώλεια σκυτάλης: Στα δίκτυα υπάρχει επίσης η πιθανότητα να χαθούν μηνύματα μεταξύ των κόμβων. Για το λόγο αυτό χρειαζόμαστε κάποια εγγύηση ότι σε κάθε χρονική στιγμή θα υπάρχει μια σκυτάλη. Ως λύση για το πρόβλημα αυτό, ορίζουμε μια από τις διεργασίες ως ερευνητής για απώλεια της σκυτάλης. Ο κόμβος αυτός εκδίδει περιοδικά ένα μήνυμα <who_has_the_token> το οποίο και στέλνει μέσα από τον λογικό δακτύλιο. Η κάθε διεργασία που λαμβάνει το μήνυμα αυτό το προωθεί στην επόμενη της, μέχρι να φτάσει πίσω στον ερευνητή. Όταν η διεργασία που το λαμβάνει έχει την σκυτάλη, αλλάζει ένα συγκεκριμένο πεδίο του μηνύματος πριν το προωθήσει. Όταν ο ερευνητής παραλάβει το μήνυμα <who_has_the_token> (μετά από μια ολόκληρη περιφορά του στον λογικό δακτύλιο) κοιτάζει το ειδικό πεδίο για να δει αν κάποια από τις διεργασίες το μετάβαλε. Αν όχι, τότε δεν υπάρχει σκυτάλη και ο ερευνητής δημιουργεί μια καινούρια. Σε αυτό το σημείο πρέπει να κάνουμε την παραδοχή ότι ο κόμβος αυτός (που ορίζεται ως ερευνητής) δεν μπορεί ποτέ να καταρρεύσει.

3.3 Προσέγγιση με Διάταξη Γεγονότων - Ο Αλγόριθμος του Lamport

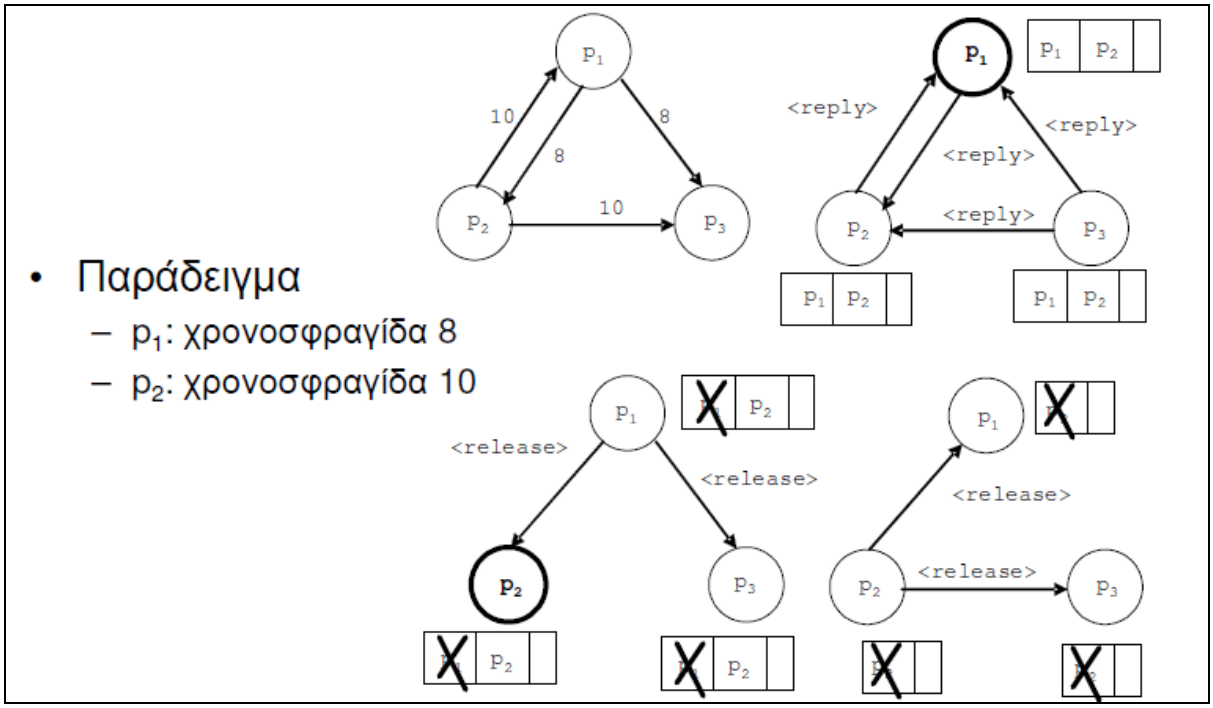
Ο αλγόριθμος του Lamport αποτελεί τον πρώτο αλγόριθμο που ακολουθεί την κατανεμημένη προσέγγιση. Γράφτηκε το 1978 και χρησιμοποιεί τις χρονοσφραγίδες Lamport (του ιδίου) για την καθολική διάταξη των γεγονότων στο σύστημα [2]. Για την υλοποίηση

του αλγορίθμου αυτού απαιτείται το σύστημα μας να είναι πλήρως συνδεδεμένο, να υπάρχει δηλαδή απευθείας επικοινωνία μεταξύ οποιωνδήποτε δυο κόμβων.

Αν μια διαδικασία p_i θέλει να εισέλθει στην κρίσιμη περιοχή
  Εισάγει την αίτηση της με μια χρονοσφραγίδα στην ουρά της
  Στέλνει ένα μήνυμα <request> με χρονοσφραγίδα t_i στις υπόλοιπες διεργασίες
Αν μια διεργασία p_j λάβει ένα μήνυμα <request> από την p_i
  Εισάγει την αίτηση της p_i στην ουρά της
  Απαντά στην p_i με ένα μήνυμα <reply>
Αν μια διεργασία p_i θέλει να εξέλθει από την κρίσιμη περιοχή
  Διαγράφει την αίτηση της από την ουρά της
  Στέλνει ένα μήνυμα <release> στις υπόλοιπες διεργασίες
Αν μια διεργασία p_j λάβει ένα μήνυμα <release> από την p_i
  Διαγράφει την αίτηση της p_i από την ουρά της
Αν μια διεργασία p_i θέλει να εισέλθει στην κρίσιμη περιοχή της
  Κοιτάζει αν έχει λάβει n-1 μηνύματα <reply> και επίσης
  κοιτάζει αν η δική της χρονοσφραγίδα είναι πιο χαμηλή από όλες
  τις υπόλοιπες χρονοσφραγίδες στην ουρά αιτήσεων
Τέλος αλγόριθμου

Σχήμα 3.3.1 Ο αλγόριθμος του Lamport

Ο αλγόριθμος του Lamport φαίνεται στο σχήμα 3.3.1. Κάθε κόμβος που επιθυμεί να εισέλθει στην κρίσιμη περιοχή στέλνει ένα μήνυμα <request> σε όλους τους κόμβους του συστήματος (συμπεριλαμβανομένου και του εαυτού του), το οποίο περιλαμβάνει και την χρονοσφραγίδα του (η χρονοσφραγίδα του περιλαμβάνει κάποιο χρονικό σημάδι για το πότε στέλνεται η αίτηση). Κάθε διεργασία διατηρεί τοπικά μια ουρά αιτήσεων, στην οποία προσθέτει τα μηνύματα αιτήσεων της κάθε διεργασίας μαζί με την ανάλογη χρονοσφραγίδα. Για να εισέλθει ο κόμβος αυτός στην κρίσιμη περιοχή θα πρέπει να συγκεντρώσει n-1 μηνύματα <reply> (η κάθε διεργασία που λαμβάνει μια αίτηση απαντά με ένα μήνυμα <reply>) και θα πρέπει επίσης η χρονοσφραγίδα του να έχει τη χαμηλότερη τιμή από όλες τις χρονοσφραγίδες που βρίσκονται στην ουρά αιτήσεων της διεργασίας. Αφού η διεργασία εισέλθει στην κρίσιμη περιοχή και τελειώσει την δουλειά της, πρέπει βγαίνοντας από την κρίσιμη περιοχή να στείλει n-1 μηνύματα <release> στις υπόλοιπες διεργασίες, οι οποίες λαμβάνοντας το μήνυμα αυτό θα αφαιρέσουν από την ουρά αιτήσεων τους την αίτηση της διεργασίας που μόλις βγήκε από την κρίσιμη περιοχή.



Σχήμα 3.3.2 Παράδειγμα χρήσης του αλγόριθμου Lamport

Ο αλγόριθμος αυτός είναι πιο πολύπλοκος από αυτόν του Lelann, για αυτό και θεώρησα πιο χρήσιμο να δείξω τη λειτουργία του χρησιμοποιώντας ένα παράδειγμα. Ας θεωρήσουμε ένα σύστημα τριών κόμβων, όπως φαίνεται στο σχήμα 3.3.2. Έστω ότι ο κόμβος p1 αποφασίζει ότι θέλει πρόσβαση στην κρίσιμη περιοχή 8 μονάδες χρόνου μετά την έναρξη της λειτουργίας του. Το ίδιο συμβαίνει και για τον κόμβο p2 αλλά 10 μονάδες χρόνου μετά την έναρξη της λειτουργίας του. Βλέπουμε στην πρώτη φάση του σχήματος 3.3.2 τις διεργασίες p1 και p2 να στέλνουν τα αντίστοιχα μηνύματα <request> με τις χρονοσφραγίδες τους. Σύμφωνα με τον αλγόριθμο, η κάθε διεργασία που λαμβάνει ένα μήνυμα <request> απαντά με ένα μήνυμα <reply> για να δείξει ότι έλαβε το μήνυμα αίτησης. Στο παράδειγμα μας, θα πρέπει να σταλούν συνολικά 4 μηνύματα <reply>, ως απάντηση για τα αντίστοιχα 4 μηνύματα <request>. Αυτό φαίνεται στην δεύτερη φάση του σχήματος 3.3.2. Επίσης φαίνονται οι ουρές αιτήσεων στον κάθε κόμβο, ταξινομημένες ως προς την μικρότερη χρονοσφραγίδα. Στο σημείο αυτό, τόσο η διεργασία p1 όσο και η διεργασία p2 έχουν λάβει n-1 μηνύματα <reply>. Όμως, η διεργασία p2 δεν μπορεί να εισέλθει στην κρίσιμη περιοχή γιατί η αίτηση της δεν βρίσκεται στην αρχή της ουράς αιτήσεων (υπάρχει δηλαδή κάποια άλλη αίτηση, στο παράδειγμα αυτό η αίτηση της p1, με μικρότερη χρονοσφραγίδα). Ως αποτέλεσμα, μόνο η διεργασία p1 θα εισέλθει στην κρίσιμη περιοχή. Αφού εκτελέσει τη λειτουργία της, βγαίνοντας από την κρίσιμη περιοχή η διεργασία p1 θα στείλει ένα μήνυμα <release> στις υπόλοιπες διεργασίες και θα αφαιρέσει την αίτηση της από την ουρά της. Λαμβάνοντας το μήνυμα <release>, και οι υπόλοιπες διεργασίες θα αφαιρέσουν την αίτηση της p1 από την ουρά τους, όπως φαίνεται στην τρίτη φάση του σχήματος 3.3.2. Αυτό έχει ως

αποτέλεσμα την ικανοποίηση της συνθήκης για την διεργασία p2, η οποία τώρα θα μπει με τη σειρά της στην κρίσιμη περιοχή. Βγαίνοντας, η διεργασία p2 θα στείλει μήνυμα <release>, και οι υπόλοιπες διεργασίες θα αφαιρέσουν και τη δική της αίτηση από την ουρά αιτήσεων τους, όπως φαίνεται στην τέταρτη φάση του σχήματος 3.3.2.

Όπως φαίνεται και από το παράδειγμα, ο αλγόριθμος αυτός ικανοποιεί και τα 3 κριτήρια υλοποίησης μιας σωστής λύσης για το πρόβλημα του αμοιβαίου αποκλεισμού. Το κριτήριο της ασφάλειας ισχύει, και αυτό αποδεικνύεται από το γεγονός ότι για να μπουν 2 διαφορετικές διεργασίες στην κρίσιμη περιοχή θα πρέπει και οι 2 να βρίσκονται στην κορυφή της ουράς αιτήσεων, κάτι το οποίο δεν γίνεται γιατί απαιτείται πρώτα να έχουν λάβει n-1 μηνύματα επιβεβαίωσης, και άρα οι 2 ουρές τους έχουν ακριβώς τα ίδια περιεχόμενα. Δεν υπάρχει πρόβλημα παρατεταμένης στέρησης γιατί οι αιτήσεις εξυπηρετούνται με τη σειρά (με βάση τις χρονοσφραγίδες τους). Για τον ίδιο λόγο ισχύει και η συνθήκη της διάταξης, αφού υπάρχει τρόπος να θέσουμε μια λογική σειρά ικανοποίησης των αιτήσεων (χρονοσφραγίδες).

Κεφάλαιο 4

Προδιαγραφή των Αλγορίθμων

4.1 Τα Αρχεία Υλοποίησης του Καναλιού MPI

Όπως προαναφέρθηκε και στο κεφάλαιο 2, τα αρχεία αυτά υλοποιήθηκαν για σκοπούς χρήσης ενός Message Passing Interface (MPI) για ανταλλαγή μηνυμάτων μεταξύ διεργασιών. Με τη χρήση τους μπορώ να υλοποιήσω ένα κανάλι επικοινωνίας για τις διεργασίες μου. Τα ολοκληρωμένα αρχεία βρίσκονται στο Παράρτημα Α ως σημεία αναφοράς. Εδώ θα περιγράψω πως υλοποιείται η επικοινωνία. Αυτό θα γίνει μέσω της επεξήγησης των μεταβάσεων που χρησιμοποιώ στις δικές μου υλοποιήσεις, καθώς και μέσω της επεξήγησης συναρτήσεων που έρχονται με τα αρχεία αυτά.

Επιγραμματικά, χρησιμοποιώντας MPI, δημιουργούμε ένα αριθμό από διεργασίες (οι οποίες στην ουσία είναι διαφορετικά νήματα που τρέχουν στον ίδιο επεξεργαστή) οι οποίες τρέχουν πανομοιότυπο κώδικα και επικοινωνούν μέσω ανταλλαγής μηνυμάτων. Η κάθε διεργασία έχει τη δική της ταυτότητα (rank) την οποία μπορεί να πάρει τοπικά εκτελώντας την συνάρτηση MPI_Rank(). Επίσης, η κάθε διεργασία γνωρίζει το πλήθος των νημάτων / διεργασιών στην συγκεκριμένη εκτέλεση, αριθμό τον οποίο μπορεί να πάρει από τη συνάρτηση MPI_Size(). Και οι δυο αυτές συναρτήσεις επιστρέφουν ένα φυσικό αριθμό (Nat). Το πλήθος των διεργασιών που θα δημιουργηθούν μπορεί να διαφέρει ανά εκτέλεση, και δίνεται ως παράμετρος γραμμής εντολών μέσω της Java (αφού πρώτα ο κώδικας μεταφραστεί). Παράδειγμα χρήσης των δύο συναρτήσεων που προαναφέρονται φαίνεται στο σχήμα 4.1.1. Στο παράδειγμα αυτό η κάθε διεργασία προσθέτει στην λίστα γειτόνων της όλες τις υπόλοιπες διεργασίες εκτός από τον εαυτό της.

```
for n : Nat where n < MPI_Size() do
        if ~(n=MPI_Rank()) then
           Nbrs:= Nbrs |- n;
        fi;
 od
```

Σχήμα 4.1.1 Παράδειγμα χρήσης των συναρτήσεων MPI

Στη συνέχεια, αφού γίνουν οι απαραίτητες αρχικοποιήσεις, χρειαζόμαστε ένα τρόπο να στέλνουμε και να λαμβάνουμε μηνύματα. Δείγμα κώδικα που χρησιμοποιεί τις εντολές αυτές δίνεται στο σχήμα 4.1.2. Σημειώνουμε πως ο κώδικας αυτός πρέπει να καλείται κάθε φορά που θέλουμε να αποστείλουμε ένα σύνολο μηνυμάτων. Για παράδειγμα, σε κάθε γύρο επανάληψης του προγράμματος μου για υλοποίηση του αλγόριθμου του Lamport απαιτείται 4 φορές η δημιουργία και αποστολή μηνυμάτων. Για το λόγο αυτό χρησιμοποίησα το τμήμα του κώδικα που φαίνεται στο σχήμα 4.1.2 τέσσερις φορές.

```
       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
        od
```

Σχήμα 4.1.2 Δείγμα εντολών για αποστολή / παραλαβή μηνυμάτων

Αφού δημιουργηθεί η προδιαγραφή των αυτομάτων και ο κώδικας μεταφραστεί σε Java (με τη χρήση του εργαλείου tempo2java [5]) μπορούμε πλέον να εκτελέσουμε τα πειράματα μας σε Java.

4.2 Προδιαγραφή του αλγόριθμου του Lamport στο εργαλείο Tempo

Αφού έχουμε κατανοήσει την γλώσσα TIOA και το μοντέλο αυτομάτων TIOA (κεφάλαιο 2) και έχουμε ορίσει πλήρως τον αλγόριθμο (κεφάλαιο 3) έχει φτάσει η στιγμή να τον υλοποιήσουμε στο εργαλείο Tempo. Όπως και στο παράδειγμα δημιουργίας (υποκεφάλαιο 2.5.) θα δημιουργήσω πρώτα ένα αυτόματο το οποίο θα περιλαμβάνει όλες τις εσωτερικές λειτουργίες ενός κόμβου που αφορούν τον αλγόριθμο του Lamport. Έπειτα, θα συνθέσω το αυτόματο αυτό με ένα κανάλι MPI για να δημιουργήσω το ολοκληρωμένο αυτόματο Lamport.

```
automaton LamportNode( u, chancetodrop, chancetoenter,timeoutwait:
DiscreteReal, rank, size: Nat)
  signature
    output SEND( m:Null[mpi_message] )
    input  RECEIVE( m:Null[mpi_message] )
    internal prepmessages ( type: Nat), init, entercritical, checkfail,
checkenter, checktimeout,timer
  states
    tosend:                      Seq[Null[mpi_message]] := { }; %the queue of
messages to be sent
    Nbrs:                Seq[Nat] := { }; % Includes all the nodes of the
system
    ackqueue:            Seq[Nat] := { }; % Stores all the nodes that sent the
ack signal
    replyqueue:                  Seq[Nat] := { }; % Stores all the nodes that
await a reply from this node
    requestqueue:        Seq[Null[mpi_message]] := { }; % Holds all the
requests of entry for all the nodes
    clock:                       DiscreteReal := 0; % The physical clock of the
automaton
    nextsend:          DiscreteReal:=0; % Time to stop the trajectory
    mytimestamp:         DiscreteReal := 0; % The timestamp
    ProcessBegan:        Bool:=true; % Indicates a working process
    criticalsection:     Bool:=false; % Indicates entry into the critical
region
    alreadyentered:      Bool:=false; % Used to avoid duplicated entry into
the critical region
    wantstoenter:        Bool:=false; % Indicates the wish to enter the
critical region
    requesting:                  Bool:=false; % Indicates that this process has
sent the request signals
    donetimer:                   DiscreteReal:=0; % A timer to tell when the
process is done with the critical region. Decided randomly.
    recovered:                   Bool:=false; % Indicates that the process has
recovered from a crash
    messagecounter:      Int:=0; % A counter to count how many messages this
node has sent
    recoverytime:        DiscreteReal:=0; % The time it will take to recover
after each failure. Decided randomly.
    recovercounter:      Int:=0; %counts how many times this process has
recovered from crash failures.
    timesentered:        Int:=0; %counts how many times this process has
entered the critical section.
    timewaited:                  DiscreteReal:=0; % Time waited from request to
entering for a single entrance into the critical section.
    maxwaited:                   DiscreteReal:=0; % Max time waited to enter.
    leastwaited:         DiscreteReal:=1000000; % Least time waited to enter.
    totalwaited:         DiscreteReal:=0; % Total time spent waiting to enter.
    messagesreceived:    Int:=0;

    initially u > 0;
```

Σχήμα 4.2.1.1 Προδιαγραφή και μεταβλητές καταστάσεων του εσωτερικού κόμβου Lamport

Στο σχήμα 4.2.1.1 φαίνεται ο ορισμός του εσωτερικού αυτόματου (LamportNode.tioa) και οι μεταβλητές καταστάσεων που περιλαμβάνει. Οι μεταβλητές καταστάσεων επεξηγούνται και στο σχήμα μέσω των σχολίων (commends - οτιδήποτε μετά το σύμβολο "%"). Δεν θα

εμβαθύνω σε αυτό το σημείο για τις μεταβλητές καταστάσεων, η λειτουργία τους θα φανεί με την επεξήγηση των μεταβάσεων. Να αναφέρω μόνο ότι η λειτουργία των Boolean μεταβλητών είναι να με βοηθήσουν στην υλοποίηση της μηχανής καταστάσεων (state machine), δηλαδή με βοηθούν να αναγνωρίσω σε ποια κατάσταση βρίσκεται ανά πάσα στιγμή ο κόμβος. Επίσης, οι τελευταίες 10 μεταβλητές δεν επηρεάζουν την λειτουργία του αυτομάτου, είναι απλά εκεί για τις μετρήσεις που θα χρειαστεί να γίνουν για την αξιολόγηση των αλγορίθμων. Θα αναφέρω μόνο τη χρήση των παραμέτρων:

- **u:** Αποτελεί τον ρυθμό εξέλιξης του αυτόματου Για παράδειγμα, για u=1 κάθε επανάληψη θα προχωρά την τροχιά μια μονάδα χρόνου (έστω ένα δευτερόλεπτο).
- **chancetodrop:** Μας δίνει την πιθανότητα (%) σε κάθε επανάληψη ο κόμβος να πέσει. Για παράδειγμα, για chancetodrop=1 ο κόμβος σε κάθε επανάληψη θα έχει 1% πιθανότητα να πάθει σφάλμα.
- **chancetoenter:** Όπως και πιο πάνω, μόνο που αυτή η παράμετρος μας δίνει την πιθανότητα για ένα κόμβο να επιθυμεί να μπει στην κρίσιμη περιοχή του.
- **timeoutwait:** Υποδεικνύει πόσες επαναλήψεις πρέπει να περάσουν μέχρι ένας κόμβος να θεωρήσει ότι οι γείτονες είναι "νεκροί".
- **size:** Είναι το σύνολο των διεργασιών που τρέχουν στο συγκεκριμένο σενάριο προσομοίωσης.
- **rank:** Αποτελεί την ταυτότητα της επεξεργασίας (processID).

Σε αυτό το σημείο να εξηγήσω πως τα μηνύματα σε αυτό τον αλγόριθμο δεν αποτελούνται από ένα αριθμό, αλλά από μια πλειάδα ενός αριθμού και μιας χρονοσφραγίδας. Ο ορισμός του τύπου MPI_message δίνεται ως εξής (όπως αλλάχτηκε στο αρχείο MPIVocabs.tioa) :

```
alg_message : Tuple [type:DiscreteReal,timestamp:AugmentedReal,sender: Nat]
tcp_message : Tuple [data:alg_message, receiver:Nat]
```

Το πεδίο timestamp αποτελεί τη χρονοσφραγίδα, ενώ το πεδίο type, το οποίο μπορεί να λάβει τιμές από 1 μέχρι 4, μας καθορίζει τον τύπο του μηνύματος ως εξής:

1. Μήνυμα Αίτησης (request message)
2. Μήνυμα Αναγνώρισης (acknowledge/reply message)
3. Μήνυμα Απελευθέρωσης (release message)
4. Μήνυμα Επαναφοράς (recovery message)

Στη συνέχεια ακολουθεί μια περιγραφή για την κάθε ενέργεια / μετάβαση. Η πλήρης προδιαγραφή του αλγόριθμου μπορεί να βρεθεί στο Παράρτημα Β.1.

- **output Send (m:Null[MPI_message]):** Η διεργασία αυτή απλά αφαιρεί το πρώτο μήνυμα από την ουρά tosend[], η οποία κρατάει όλα τα προς αποστολή μηνύματα. Επίσης , αυξάνει το μετρητή μηνυμάτων messagecounter κατά 1 κάθε φορά που εκτελείται. Η παράμετρος m είναι τύπου Null[MPI_message] το οποίο σημαίνει πως μπορεί να μεταβληθεί η τιμή της από την κλήση της ενέργειας αυτής (αντίστοιχα με τους δείκτες και την κλήση συναρτήσεων με αναφορά στην γλώσσα C).
- **input Receive (m:Null[MPI_message]):** Η μέθοδος αυτή λαμβάνει ένα μήνυμα. Ανάλογα με τον τύπο του μηνύματος (που καθορίζεται από το πεδίο type, το οποίο είναι με τη σειρά του υποπεδίο του πεδίου data του m) γίνεται και ο αντίστοιχος χειρισμός ως εξής:
    1. Σηματοδοτεί μήνυμα αίτησης. Στην περίπτωση αυτή προσθέτουμε ολόκληρο το μήνυμα στην ουρά αιτήσεων, ούτως ώστε να ξέρουμε ποιος είναι ο κόμβος που αιτείται (m.data.sender) και ποια είναι η χρονοσφραγίδα του (m.data.timestamp). Επίσης, προσθέτουμε τον αποστολέα στην ουρά απαντήσεων ούτως ώστε να γνωρίζουμε ότι ο αποστολέας περιμένει επιβεβαίωση από εμάς.
    2. Σηματοδοτεί μήνυμα αναγνώρισης. Απλά τοποθετούμε τον αποστολέα στην ουρά αναγνώρισης (replyqueue) για να γνωρίζουμε πως ο κόμβος αυτός έχει λάβει το μήνυμα μας και έχει απαντήσει.
    3. Σηματοδοτεί μήνυμα απελευθέρωσης. Όταν μια διεργασία λάβει αυτό το μήνυμα σημαίνει πως ο αποστολέας έχει μόλις βγει από την κρίσιμη περιοχή. Ως αποτέλεσμα, θα πρέπει να εξάγει την αίτηση του αποστολέα από την ουρά αιτήσεων.
    4. Σηματοδοτεί μήνυμα επαναφοράς. Με τη λήψη αυτού του μηνύματος η διεργασία γνωρίζει πως ο αποστολέας είχε πάθει σφάλμα και μόλις επανήλθε σε λειτουργία. Για το λόγο αυτό προσθέτει τον κόμβο στη λίστα με τους γείτονες (Nbrs) αν δεν υπάρχει ήδη εκεί.
- **internal init:** Απλά αρχικοποιεί τη λίστα με τους γείτονες (Nbrs) που περιέχει όλους τους συμμετέχοντες κόμβους εκτός του εαυτού της κάθε διεργασίας.
- **internal timer:** Απλά υπολογίζει την τιμή της μεταβλητής nextsend, η οποία χρησιμοποιείται για την εξέλιξη της τροχιάς κατά "u" βήματα σε κάθε επανάληψη (όπως εξηγείται και αργότερα στην επεξήγηση της τροχιάς).

- **internal checkfail:** Αποφασίζει τυχαία αν η διεργασία θα πάθει σφάλμα. Η πιθανότητα να πάθει σφάλμα σε κάθε επανάληψη δίνεται ως παράμετρος στο αυτόματο. Στην περίπτωση που πάθει σφάλμα, θέτω την μεταβλητή καταστάσεων ProcessBegan σε "false" και ορίζω ένα τυχαίο χρόνο στον οποίο η διεργασία θα επανέλθει σε λειτουργία (μεταξύ 10 και 40 επαναλήψεων μετά τη βλάβη). Αποθηκεύω το χρόνο αυτό στην μεταβλητή καταστάσεων recoverytime. Επίσης, είναι υπεύθυνη για την επαναφορά της διεργασίας όταν το ρολόι φτάσει το recoverytime, στην οποία περίπτωση επαναφέρει την μεταβλητή καταστάσεων ProcessBegan σε "true" και θέτει και την μεταβλητή κατστάσεων recovered σε "true". Τέλος, αυξάνει τον μετρητή recoveredcounter κατά 1.
- **internal checkenter:** Αποφασίζει τυχαία αν η διεργασία θέλει να μπει στην κρίσιμη περιοχή της. Η πιθανότητα να θέλει η διεργασία σε κάθε επανάληψη να εισέλθει στην κρίσιμη περιοχή δίνεται ως παράμετρος. Σε περίπτωση που αποφασιστεί ότι η διεργασία θέλει να μπει στην κρίσιμη περιοχή, θέτουμε ως "true" την μεταβλητή καταστάσεων wantstoenter. Επίσης, θέτουμε την τιμή της μεταβλητής mytimestamp ως την τιμή του ρολογιού, ούτως ώστε να έχουμε την χρονοσφραγίδα μας. Σημειώνεται πως εφόσον μια διεργασία θέλει να μπει στην κρίσιμη περιοχή δεν μπορεί να ξαναζητήσει να μπει μέχρις ότου ικανοποιηθεί η πρώτη αίτηση. Ο περιορισμός αυτός αποφεύγει τα προβλήματα παρατεταμένης στέρησης.
- **internal checktimeout:** Αποφασίζει αν ορισμένοι κόμβοι του δικτύου πρέπει να θεωρηθούν ως εσφαλμένοι και να αφαιρεθούν από τη λίστα των γειτόνων. Πυροδοτείται "timeoutwait" κύκλους (όπως δίνεται στις παραμέτρους) μετά την αίτηση της διεργασίας να μπει στην κρίσιμη περιοχή. Αν δεν λάβει μήνυμα <acknowledge> από κάποια διεργασία την θεωρεί "νεκρή" και την αφαιρεί από την λίστα Nbrs.
- **internal prepmessages(type:Nat):** Προετοιμάζει τα προς αποστολή μηνύματα. Η παράμετρος type καθορίζει τι τύπου μηνύματα πρέπει να προετοιμαστούν. Η μετάβαση αυτή πρέπει να καλεστεί 4 φορές σε κάθε επανάληψη, μια φορά για κάθε τύπο μηνύματος. Δεν εκτελείται όταν η διεργασία δεν είναι ενεργή (ProcessBegan = false) ή όταν η ουρά tosend δεν είναι κενή (υπάρχουν δηλαδή ακόμη μηνύματα προς αποστολή). Η εκτέλεση της ανάλογα με την παράμετρο type περιγράφεται ως εξής:

1. Προυποθέτει ότι η διεργασία θέλει να μπει στην κρίσιμη περιοχή (wantstoenter = true) αλλά δεν έχει στείλει ακόμη της αιτήσεις της (requesting = false). Γεμίζει την ουρά αποστολής μηνυμάτων tosend με ένα μήνυμα <request> για κάθε κόμβο στην λίστα γειτόνων και θέτει την μεταβλητή καταστάσεων requesting σε "true" για να αποφύγουμε διπλή αποστολή.

2. Βάζει ένα μήνυμα <acknowledge> στην ουρά αποστολής μηνυμάτων για κάθε κόμβο που βρίσκεται στο στη λίστα αναμονής απάντησης replyqueue και στη συνέχεια αδειάζει τα περιεχόμενα της ουράς replyqueue. Ένας κόμβος μπορεί να βρίσκεται στην ουρά replyqueue μόνο όταν έχει στείλει μήνυμα αίτησης, όπως φαίνεται και στον ορισμό της μετάβασης receive.
3. Πυροδοτείται μόνο όταν ο κόμβος βρίσκεται στην κρίσιμη περιοχή και έχει τελειώσει την δουλειά του (criticalsection /\ clock >= donetimer). Στην περίπτωση αυτή εξέρχεται από την κρίσιμη περιοχή (criticalsection=false και alreadyentered=false) και βάζει ένα μήνυμα <release> για κάθε ζωντανό γείτονα στην ουρά αποστολής μηνυμάτων.
4. Πυροδοτείται μόνο όταν η μεταβλητή recovered έχει την τιμή "true", δηλαδή η διεργασία έχει μόλις ανακάμψει από σφάλμα. Στην περίπτωση αυτή δημιουργεί ένα μήνυμα <recovery> για κάθε ζωντανή διεργασία και το εισάγει στην ουρά μηνυμάτων προς αποστολή.

- **internal entercritical:** Η μετάβαση αυτή βάζει τη διεργασία στην κρίσιμη περιοχή αφού ελέγξει πρώτα ότι ικανοποιεί τις 2 βασικές συνθήκες του αλγόριθμου (έχει λάβει σήμα <acknowledge> από όλους τους ενεργούς γείτονες και η δική της χρονοσφραγίδα είναι η πιο μικρή). Στην περίπτωση αυτή θέτει την μεταβλητή criticalsection σε "true" και τις μεταβλητές wantstoenter και requesting σε "false". Τέλος, αποφασίζει ένα τυχαίο χρόνο στον οποίο θα μείνει εντός της κρίσιμης περιοχής (από 1 ως 11 κύκλους), ενημερώνει ανάλογα την μεταβλητή donetimer και υπολογίζει τις στατιστικές τιμές για τις μεταβλητές καταστάσεων totalwaited, timesentered, leastwaited και maxwaited.

Τελειώνοντας, η τροχιά του αυτόματου αυτού εξελίσσεται με ρυθμό u, το οποίο δίνεται από τις παραμέτρους του αυτομάτου ("evolve d(clock)=1 και stop when clock>=nextsend"). Τυπικά, το u μπορεί να έχει πάντα την τιμή 1, αλλά είναι σωστή πρακτική να δίνουμε την δυνατότητα στον χρήστη να μεταβάλει τον ρυθμό αύξησης της τροχιάς.

4.2.2 Ο ολοκληρωμένος κόμβος Lamport

Αφού ορίσαμε τον εσωτερικό κόμβο Lamport μπορούμε τώρα να ορίσουμε τον ολοκληρωμένο κόμβο Lamport (LamportSystem.tioa), ο οποίος αποτελείται από ένα αυτόματο τύπου LamportNode και τα αυτόματα που χρειάζονται για την υλοποίηση του καναλιού MPI. Η σύνθεση και ο ορισμός των μεταβλητών καταστάσεων του scheduler φαίνεται στο σχήμα 4.2.2.1.

```
automaton LamportSystem(u,chancetodrop,chancetoenter,timeoutwait: DiscreteReal,
runs:Nat)
  components
    LN : LamportNode(u,chancetodrop,chancetoenter,timeoutwait,MPI_Rank(),
MPI_Size());
    SM : SendMediator;
       RM : ReceiveMediator;

schedule
states
   m : Null[mpi_message] := nil();
   request   :Nat:=1;
   ack       :Nat:=2;
   release   :Nat:=3;
   recovery  :Nat:=4;
```

Σχήμα 4.2.2.1 Ορισμός και μεταβλητές σύνθεσης

Αφού υλοποιηθεί η σύνθεση και δηλώσουμε τις απαραίτητες μεταβλητές, φτάνουμε στο κεντρικό σημείο του προγράμματος (προσομοίωσης). Εδώ θα ορίσουμε ρητά τη ροή εκτέλεσης των μεταβάσεων. Τμήμα της ροής αυτής φαίνεται στο σχήμα 4 2 2.2 . Όπως φαίνεται , σε κάθε επανάληψη του αλγόριθμου ο κόμβος / διεργασία αρχικά ελέγχει την πιθανότητα να πάθει σφάλμα (ή αν βρίσκεται ήδη σε λανθάνουσα κατάσταση θα ελέγξει αν έχει έρθει η ώρα για επαναφορά). Αφού περάσει από το στάδιο αυτό, θα προσπαθήσει (αν δεν έχει πάθει σφάλμα) να δει αν θέλει να μπει στην κρίσιμη περιοχή (ή αν βρίσκεται ήδη στην κρίσιμη περιοχή και έχει περάσει ο απαραίτητος χρόνος, να θέσει τη μεταβλητή done σε "true", σηματοδοτώντας έτσι ότι η διεργασία έχει τελειώσει από την κρίσιμη περιοχή). Αφού περάσει και από αυτό το στάδιο, θα δει αν κάποια διεργασία έχει πάθει σφάλμα (άργησε πάνω από "timeoutwait" κύκλους να απαντήσει) και θα την διαγράψει από την λίστα των γειτόνων της. Ακολούθως, πρέπει να τρέξουμε τις ενέργειες prepmessages, send και receive τέσσερις φορές, μια για κάθε τύπο μηνύματος. Στο σχήμα φαίνεται παράδειγμα εκτέλεσης για τα μηνύματα αιτήσεων. Ο ίδιος κώδικας πρέπει να αντιγραφεί τέσσερις φορές, μια για κάθε τύπο μηνύματος (αλλάζοντας την παράμετρο εισόδου της ενέργειας αλλάζει και ο ανάλογος τύπος μηνύματος). Τελικά, σε κάθε επανάληψη εκτελείται η ενέργεια entercritical, η οποία ελέγχει κατά πόσο η διεργασία έχει το δικαίωμα να μπει στην κρίσιμη περιοχή. Αν θα μπει τότε καθορίζει ένα τυχαίο χρονικό διάστημα (1-11 επαναλήψεις) κατά το οποίο θα παραμείνει εντός της κρίσιμης περιοχής και ενημερώνει τις μεταβλητές καταστάσεων που την αφορούν, όπως αυτές περιγράφονται πιο πάνω.

```
do

   fire internal LN.init;
   % -- From here starts the main loop of the execution
   for i : Nat where i < runs do
      follow LN.T duration \infty;

        % -- Fix the timer of this run
      fire internal LN.timer;

        % -- Check if the process will randomly fail
      fire internal LN.checkfail;

      % -- Check if the process randomly wants to enter the critical region
      fire internal LN.checkenter;

      % -- Check if some nodes timed out
      fire internal LN.checktimeout;


       % -- Prepare the request signals
      fire internal LN.prepmessages(request);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
         od
```

Σχήμα 4.2.2.2 Τμήμα του προγράμματος υλοποίησης

4.3 Προδιαγραφή του αλγόριθμου του Lelann στο εργαλείο Tempo

Στο σημείο αυτό θα ορίσω το μοντέλο TIOA που δημιούργησα για να προσομοιώσω τις λειτουργίες του αλγόριθμου του Lelann σε γλώσσα TIOA. Και πάλι όρισα το εσωτερικό αυτόματο (LelannNode.tioa) του οποίου η πλήρης προδιαγραφή φαίνεται στο Παράρτημα Β.3. Το αυτόματο αυτό εκτελεί όλες τις εσωτερικές λειτουργίες που χρειάζονται για υλοποίηση του αλγόριθμου του Lelann. Για να ολοκληρωθεί όμως το σύστημα μας χρειαζόμαστε και τα αυτόματα για το κανάλι, τα οποία θα συνθέσουμε με το εσωτερικό αυτόματο για να πάρουμε την τελική μορφή ενός πλήρους αυτόματου το οποίο θα μεταφραστεί σε Java.

Στο σημείο αυτό αξίζει να αναφέρω πως δεν υλοποίησα την δεύτερη περίπτωση σφαλμάτων κατά την οποία χάνεται η σκυτάλη, και αυτό γιατί κάτι τέτοιο θα ανάγκαζε τον αλγόριθμο να

ξεφύγει από την κατανεμημένη προσέγγιση. Αυτό θα συνέβαινε γιατί ο κόμβος που θα εκλεγόταν ως αρχηγός θα έπρεπε να μην πάθει ποτέ σφάλμα, κάτι το οποίο στον πραγματικό κόσμο είναι αδύνατο, και επίσης θα έπρεπε να εκτελεί διαφορετικό κώδικα από τους υπόλοιπους κόμβους.

4.3.1 Ο εσωτερικός κόμβος Lelann (LelannNode.tioa)

Στο σχήμα 4.3.1.1 φαίνεται ο ορισμός του εσωτερικού αυτόματου Lelann, καθώς και οι μεταβλητές καταστάσεων που χρειάζεται. Δεν θα αναλύσω τις παραμέτρους που παίρνει, μιας και είναι οι ίδιες με τον αλγόριθμο Lamport. Θα αναλύσω ωστόσο τις μεταβάσεις που το αποτελούν. Ο ολοκληρωμένος κώδικας για αυτό το αυτόματο βρίσκεται στο Παράρτημα Β.3.

Σε αυτό το σημείο να εξηγήσω πως τα μηνύματα σε αυτό τον αλγόριθμο δεν αποτελούνται από ένα αριθμό, αλλά από μια πλειάδα ενός αριθμού και ενός ζευγαριού αποστολέα / παραλήπτη. Επίσης, η λίστα γειτόνων δεν αποτελείται πλέον από διεργασίες, αλλά από πλειάδες τύπου process_status, οι οποίες χρειάζονται για περιπτώσεις βλάβης σε συγκεκριμένες διεργασίες. Ο ορισμός του τύπου process_status καθώς και του τύπου mpi_message δίνεται ως εξής (όπως προστέθηκε / αλλάχτηκε στο αρχείο MPIVocabs.tioa) :

```
process_status : Tuple [rank:Nat, online:Bool]
alg_message : Nat
MPI_message : Tuple [data:alg_message, sender:Node, receiver:Node]
```

Το πεδίο data, το οποίο μπορεί να λάβει τιμές από 1 μέχρι 3, μας καθορίζει τον τύπο του μηνύματος ως εξής:

1. Μήνυμα σκυτάλη (token message)
2. Μήνυμα Αναγνώρισης (acknowledge/reply message)
3. Μήνυμα Επαναφοράς (recovery message)

Στη συνέχεια ακολουθεί μια περιγραφή για την κάθε ενέργεια / μετάβαση. Η πλήρης προδιαγραφή του αλγόριθμου μπορεί να βρεθεί στο Παράρτημα Β.3.

- **output Send (m:Null[mpi_message]):** Η διεργασία αυτή απλά αφαιρεί το πρώτο μήνυμα από την ουρά tosend[], η οποία κρατάει όλα τα προς αποστολή μηνύματα. Επίσης , αυξάνει το μετρητή μηνυμάτων messagecounter κατά 1 κάθε φορά που εκτελείται. Η παράμετρος m είναι τύπου Null[mpi_message] το οποίο σημαίνει πως

μπορεί να μεταβληθεί η τιμή της από την κλήση της ενέργειας αυτής (αντίστοιχα με τους δείκτες και την κλήση συναρτήσεων με αναφορά στην γλώσσα C).

- **input Receive (m:Null[mpi_message]):** Η μέθοδος αυτή λαμβάνει ένα μήνυμα. Ανάλογα με τον τύπο του μηνύματος (που καθορίζεται από το πεδίο data του m) γίνεται και ο αντίστοιχος χειρισμός όπως θα περιγραφεί στη συνέχεια. Σημειώνουμε επίσης πως σε κάθε περίπτωση κλήσης αυτής της ενέργειας, η μεταβλητή prevavailable παίρνει την τιμή της διεύθυνσης του αποστολέα (val(m).sender). Αυτό γίνεται για να γνωρίζουμε ανά πάσα στιγμή σε ποιά διεργασία θα πρέπει να απαντήσουμε με μήνυμα αναγνώρισης, για να μπορεί η διεργασία αυτή να γνωρίζει ότι είμαστε ακόμη ενεργοί.
    1. Σηματοδοτεί το μήνυμα σκυτάλη. Λαμβάνοντας το θέτουμε την τιμή της μεταβλητής token (η οποία αντιπροσωπεύει την κατοχή της σκυτάλης) σε "true". Επίσης, προσθέτουμε τον αποστολέα στην ουρά απαντήσεων και θέτουμε την τιμή της μεταβλητής mustreply σε "true", ούτως ώστε να γνωρίζουμε ότι ο αποστολέας περιμένει επιβεβαίωση από εμάς.
    2. Σηματοδοτεί μήνυμα αναγνώρισης. Απλά θέτουμε την τιμή της μεταβλητής requestingack σε "false", αφού πλέον έχουμε λάβει το μήνυμα αναγνώρισης.
    3. Σηματοδοτεί μήνυμα επαναφοράς. Με τη λήψη αυτού του μηνύματος η διεργασία βρίσκει τον αποστολέα στη λίστα γειτόνων και θέτει το πεδίο online της συγκεκριμένης διαδικασίας σε "true".
- **internal init:** Απλά αρχικοποιεί τη λίστα με τους γείτονες (Nbrs) που περιέχει όλους τους συμμετέχοντες κόμβους. Επίσης δίνει τη σκυτάλη στην διεργασία μηδέν, ούτως ώστε να μπορεί να αρχίσει η εκτέλεση του αλγόριθμου. Η διεργασία αυτή καλείται μόνο μια φορά στην αρχή της εκτέλεσης του σεναρίου.
- **internal timer:** Απλά υπολογίζει την τιμή της μεταβλητής nextsend, η οποία χρησιμοποιείται για την εξέλιξη της τροχιάς κατά "u" βήματα σε κάθε επανάληψη (όπως εξηγείται και αργότερα στην επεξήγηση της τροχιάς).
- **internal updatetopology:** Απλά βρίσκει την επόμενη διαθέσιμη διεργασία στο σύστημα.
- **internal checkfail:** Αποφασίζει τυχαία αν η διεργασία θα πάθει σφάλμα. Η πιθανότητα να πάθει σφάλμα σε κάθε επανάληψη δίνεται ως παράμετρος στο αυτόματο. Στην περίπτωση που πάθει σφάλμα, θέτω τις μεταβλητές καταστάσεων ProcessBegan, token και wantstoenter σε "false" και ορίζω ένα τυχαίο χρόνο στον οποίο η διεργασία θα επανέλθει σε λειτουργία (μεταξύ 10 και 40 επαναλήψεων μετά τη βλάβη). Αποθηκεύω το χρόνο αυτό στην μεταβλητή καταστάσεων recoverytime. Επίσης, είναι υπεύθυνη για την επαναφορά της διεργασίας όταν το ρολόι φτάσει το

recoverytime, στην οποία περίπτωση επαναφέρει την μεταβλητή καταστάσεων ProcessBegan σε "true" και θέτει και την μεταβλητή κατστάσεων recovered σε "true". Τέλος, αυξάνει τον μετρητή recovercounter κατά 1.

- **internal checkenter:** Αποφασίζει τυχαία αν η διεργασία θέλει να μπει στην κρίσιμη περιοχή της. Η πιθανότητα να θέλει η διεργασία σε κάθε επανάληψη να εισέλθει στην κρίσιμη περιοχή δίνεται ως παράμετρο ς Σε περίπτωση που αποφασιστεί ότι η διεργασία θέλει να μπει στην κρίσιμη περιοχή, θέτουμε ως "true" την μεταβλητή καταστάσεων wantstoenter. Επίσης, θέτουμε την τιμή της μεταβλητής timerequested ως την τιμή του ρολογιού, ούτως ώστε να έχουμε κάποιο τρόπο να υπολογίσουμε τον χρόνο αναμονής της διεργασίας μέχρι την είσοδο της στην κρίσιμη περιοχή.
- **internal checktimeout:** Στην μετάβαση αυτή ο συγκεκριμένος κόμβος αποφασίζει αν ο επόμενος του πρέπει να θεωρηθεί "νεκρός". Πυροδοτείται όταν έχουν περάσει "timeoutwait" επαναλήψεις (όπως αυτό ορίζεται από την παράμετρο) από την ώρα που η διεργασία αυτή έστειλε κάποιο μήνυμα και δεν έχει λάβει ακόμη απάντηση. Βρίσκει τον επόμενο της κόμβο από τη λίστα των γειτόνων και μεταβάλλει την τιμή του πεδίου online του συγκεκριμένου κόμβου σε "false". Επίσης δημιουργεί μια νέα σκυτάλη προς αποστολή.
- **internal prepmessages(type:Nat):** Προετοιμάζει τα προς αποστολή μηνύματα. Η παράμετρος type καθορίζει τι τύπου μηνύματα πρέπει να προετοιμαστούν. Η μετάβαση αυτή πρέπει να καλεστεί 3 φορές σε κάθε επανάληψη, μια φορά για κάθε τύπο μηνύματος. Δεν εκτελείται όταν η διεργασία δεν είναι ενεργή (ProcessBegan = false) ή όταν η ουρά tosend δεν είναι κενή (υπάρχουν δηλαδή ακόμη μηνύματα προς αποστολή). Η εκτέλεση της ανάλογα με την παράμετρο type περιγράφεται ως εξής:

1. Προωθεί την σκυτάλη στην επόμενη διαθέσιμη διεργασία (nextavailable) όταν έχει την σκυτάλη (token-true) και δεν την θέλει (wantstoenter=false) ούτε βρίσκεται στην κρίσιμη περιοχή της (criticalsection=false). Επίσης θέτει την τιμή της μεταβλητής requestingack σε "true", ορίζοντας έτσι ότι αναμένει σήμα αναγνώρισης (χρησιμεύει στα timeouts). Τέλος, θέτει την τιμή της μεταβλητής "timerequested" σε clock.
2. Πυροδοτείται όταν η μεταβλητή mustreply έχει την τιμή "true". Προσθέτει στην ουρά αναμονής μηνυμάτων το μήνυμα <acknowledge> προς τον κόμβο prevavailable (δηλαδή τον κόμβο που του έστειλε τελευταίος μήνυμα).
3. Πυροδοτείται μόνο όταν η μεταβλητή recovered έχει την τιμή "true", δηλαδή η διεργασία έχει μόλις ανακάμψει από σφάλμα. Στην περίπτωση αυτή δημιουργεί ένα μήνυμα <recovery> για την προηγμούμενη ζωντανή διεργασία (prevavailable).

```
automaton LelannNode(u, chancetodrop, chancetoenter,timeoutwait: DiscreteReal)
where u > 0
  signature
    output SEND( m:Null[mpi_message] )
    input  RECEIVE( m:Null[mpi_message] )
    internal prepmessages(type:Nat), init,entercritical,checkfail, checkenter,
checktimeout, updatetopology


  states

    entered: AugmentedReal:=0;
    tosend:  Seq[Null[mpi_message]] := { }; %the queue of messages to be sent
    Nbrs:    Seq[process_status] := { }; % Includes all the nodes of the system
    index:   Nat:=0;
    nextindex: Nat:=0;
    nextavailable:  Nat:=0;
    prevavailable:  Nat:=0;
    clock:   AugmentedReal := 0; % The physical clock of the automaton
    timerequested: AugmentedReal := 0; % The time requested to enter in the
critical section.
    ProcessBegan:   Bool:=true; % Indicates a working process
    criticalsection:Bool:=false; % Indicates entry into the critical region
    token:          Bool:=false; %Indicates that the process has the token
    wantstoenter:   Bool:=false; % Indicates the wish to enter the critical
region
    mustreply:      Bool:=false; % Indicates that this process has to send a
reply message
    requestingack:  Bool:=false; % Indicates that the process is waiting for an
ack signal
    done:    Bool:=false; % Indicates that the process is done with the
critical region
    donetimer:AugmentedReal:=0; % A timer to tell when the process is done with
the critical region. Decided randomly.
    recovered: Bool:=false; % Indicates that the process has recovered from a
crash
    messagecounter: AugmentedReal:=0; % A counter to count how many messages
this node has sent
    recoverytime:   AugmentedReal:=0; % The time it will take to recover after
each failure. Decided randomly.
    recovercounter: Int:=0; %counts how many times this process has recovered
from crash failures.
    timesentered:   Int:=0; %counts how many times this process has entered the
critical section.
    timewaited: AugmentedReal:=0; % Time waited from request to entering for a
single entrance into the critical section.
    maxwaited:      AugmentedReal:=0; % Max time waited to enter.
    leastwaited:          AugmentedReal:=1000000; % Least time waited to enter.
    totalwaited:    AugmentedReal:=0; % Total time spent waiting to enter.
```

Σχήμα 4.3.1.1 Προδιαγραφή και Μεταβλητές καταστάσεων του εσωτερικού κόμβου Lelann

- **internal entercritical:** Η μετάβαση αυτή βάζει τη διεργασία στην κρίσιμη περιοχή αφού ελέγξει πρώτα ότι ικανοποιεί τις 2 βασικές συνθήκες το υ αλγόριθμου (έχει λάβει την σκυτάλη και θέλει να μπει στην κρίσιμη περιοχή). Στην περίπτωση αυτή θέτει την μεταβλητή criticalsection σε "true" και την μεταβλητή wantstoenter σε "false". Επίσης αποφασίζει ένα τυχαίο χρόνο στον οποίο θα μείνει εντός της κρίσιμης περιοχής (από 1 ως 11 κύκλους), ενημερώνει ανάλογα την μεταβλητή donetimer και

υπολογίζει τις στατιστικές τιμές για τις μεταβλητές καταστάσεων totalwaited, timesentered, leastwaited και maxwaited.). Τέλος, είναι υπεύθυνη να βγάλει τη διεργασία από την κρίσιμη περιοχή όταν αυτή τελειώσει τη δουλειά της (clock >= donetimer) και σηματοδοτεί ότι έχει τελειώσει από την κρίσιμη περιοχή (done = true).

Τελειώνοντας, η τροχιά του αυτόματου αυτού εξελίσσεται με ρυθμό u, το οποίο δίνεται από τις παραμέτρους του αυτομάτου ("evolve d(clock)=1 και stop when clock>=nextsend"). Τυπικά, το u μπορεί να έχει πάντα την τιμή 1, αλλά είναι σωστή πρακτική να δίνουμε την δυνατότητα στον χρήστη να μεταβάλει τον ρυθμό αύξησης της τροχιάς.

4.3.2 Ο ολοκληρωμένος κόμβος Lelann

Αφού ορίσαμε τον εσωτερικό κόμβο Lelann μπορούμε τώρα να ορίσουμε τον ολοκληρωμένο κόμβο Lelann (LelannSystem.tioa), ο οποίος αποτελείται από ένα αυτόματο τύπου LelannNode και τα αυτόματα που χρειάζονται για την υλοποίηση του καναλιού MPI. Η σύνθεση και ο ορισμός των μεταβλητών καταστάσεων του scheduler φαίνεται στο σχήμα 4.3.2.1. Η πλήρης προδιαγραφή του υπάρχει στο Παράρτημα Β.4.

```
automaton LelannSystem(u,chancetodrop,chancetoenter,timeoutwait: DiscreteReal,
runs:Nat)
  components
    LN : LelannNode(u,chancetodrop,chancetoenter,timeoutwait);
    SM : SendMediator;
    RM : ReceiveMediator;

schedule

states
   m : Null[mpi_message] := nil();
   request   :Nat:=1;
   ack       :Nat:=2;
   recovery  :Nat:=3;
```

Σχήμα 4.3.2.1 Ορισμός και μεταβλητές σύνθεσης

Αφού υλοποιηθεί η σύνθεση και δηλώσουμε τις απαραίτητες μεταβλητές, φτάνουμε στο κεντρικό σημείο του προγράμματος (προσομοίωσης). Εδώ θα ορίσουμε ρητά τη ροή εκτέλεσης των μεταβάσεων. Τμήμα της ροής αυτής φαίνεται στο σχήμα 4 3.2.2. Όπως φαίνεται , σε κάθε επανάληψη του αλγόριθμου ο κόμβος / διεργασία αρχικά ελέγχει την πιθανότητα να πάθει σφάλμα (ή αν βρίσκεται ήδη σε λανθάνουσα κατάσταση θα ελέγξει αν έχει έρθει η ώρα για επαναφορά). Αφού περάσει από το στάδιο αυτό, θα προσπαθήσει (αν δεν έχει πάθει σφάλμα) να δει αν θέλει να μπει στην κρίσιμη περιοχή (ή αν βρίσκεται ήδη στην κρίσιμη περιοχή και έχει περάσει ο απαραίτητος χρόνος, να θέσει τη μεταβλητή done σε "true", σηματοδοτώντας έτσι ότι η διεργασία έχει τελειώσει από την κρίσιμη περιοχή). Αφού

περάσει και από αυτό το στάδιο, θα δει αν κάποια διεργασία έχει πάθει σφάλμα (άργησε πάνω από "timeoutwait" κύκλους να απαντήσει) και θα την διαγράψει από την λίστα των γειτόνων της. Μετά καλεί την ενέργεια updatetopology για να γνωρίζει την αμέσως επόμενη "ζωντανή" διεργασία. Ακολούθως, πρέπει να τρέξουμε τις ενέργειες prepmessages, send και receive τρεις φορές, μια για κάθε τύπο μηνύματος. Στο σχήμα φαίνεται παράδειγμα εκτέλεσης για τα μηνύματα αιτήσεων. Ο ίδιος κώδικας πρέπει να αντιγραφεί τρεις φορές, μια για κάθε τύπο μηνύματος (αλλάζοντας την παράμετρο εισόδου της ενέργειας αλλάζει και ο ανάλογος τύπος μηνύματος). Τελικά, σε κάθε επανάληψη εκτελείται η ενέργεια entercritical, η οποία ελέγχει κατά πόσο η διεργασία έχει το δικαίωμα να μπει στην κρίσιμη περιοχή. Αν θα μπει τότε καθορίζει ένα τυχαίο χρονικό διάστημα (1-11 επαναλήψεις) κατά το οποίο θα παραμείνει εντός της κρίσιμης περιοχής και ενημερώνει τις μεταβλητές καταστάσεων που την αφορούν, όπως αυτές περιγράφονται πιο πάνω.

```
do
   fire internal LN.init;
   % -- From here starts the main loop of the execution
   for i : Nat where i < runs do
      follow LN.T duration \infty;

        % -- Fix the timer of this run
      fire internal LN.timer;

        % -- Check if the process will randomly fail
      fire internal LN.checkfail;

      % -- Check if the process randomly wants to enter the critical region
      fire internal LN.checkenter;

      % -- Check if some nodes timed out
      fire internal LN.checktimeout;

      % -- update the topology of the network
      fire internal LN.updatetopology;

       % -- Prepare the request signals
      fire internal LN.prepmessages(request);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
      od
```

Σχήμα 4.3.2.2 Τμήμα του προγράμματος υλοποίησης

Κεφάλαιο 5

Πειραματική Αξιολόγηση

5.1 Η Βιβλιοθήκη MPJ Express

Για να αξιολογήσω τους αλγόριθμους που υλοποίησα χρησιμοποίησα την βιβλιοθήκη ανοιχτού πηγαίου κώδικα MPJ Express, η οποία διανέμεται υπό την αιγίδα του MIT [8]. Για να χρησιμοποιήσει κάποιος τη βιβλιοθήκη αυτή πρέπει πρώτα να την εγκαταστήσει. Το πλήρες πακέτο της βιβλιοθήκης βρίσκεται στην επίσημη ιστοσελίδα της MPJ Express [8]. Αφού γίνει η εγκατάσταση, σύμφωνα πάντα με τις οδηγίες που παρέχονται, μπορεί να χρησιμοποιηθεί είτε τοπικά σε ένα υπολογιστή, είτε σε clusters. Η βιβλιοθήκη αυτή επιτρέπει την ανάπτυξη και εκτέλεση παράλληλων προγραμμάτων στην γλώσσα Java, τα οποία υλοποιούνται με τη δημιουργία διαφορετικών νημάτων επεξεργασίας (threads) ανά διεργασία. Οι διεργασίες αυτές επικοινωνούν μεταξύ τους μέσω ανταλλαγής μηνυμάτων. Για να γίνει αυτό απαιτείται μια διαφάνεια ανταλλαγής μηνυμάτων (Message Passing Interface - MPI).

Η βιβλιοθήκη αυτή υπάρχει τόσο για το λειτουργικό σύστημα Windows όσο και για Unix-based λειτουργικά συστήματα (π.χ. Linux). Για τα δικά μου σενάρια προσομοίωσης χρησιμοποίησα λειτουργικό σύστημα Windows 7 και την διαφάνεια εργασίας Eclipse. Για να ρυθμιστεί η επιφάνεια εργασίας Eclipse να υποστηρίζει προγράμματα που χρησιμοποιούν τη βιβλιοθήκη MPJ Express υπάρχουν χρήσιμες οδηγίες στην επίσημη ιστοσελίδα της MPJ Express.

Για να τρέξουμε ένα τέτοιο πρόγραμμα στην πλατφόρμα Eclipse θα πρέπει να ορίσουμε τις εξής παραμέτρους προγράμματος και ιδεατής μηχανής (Virtual Machine):

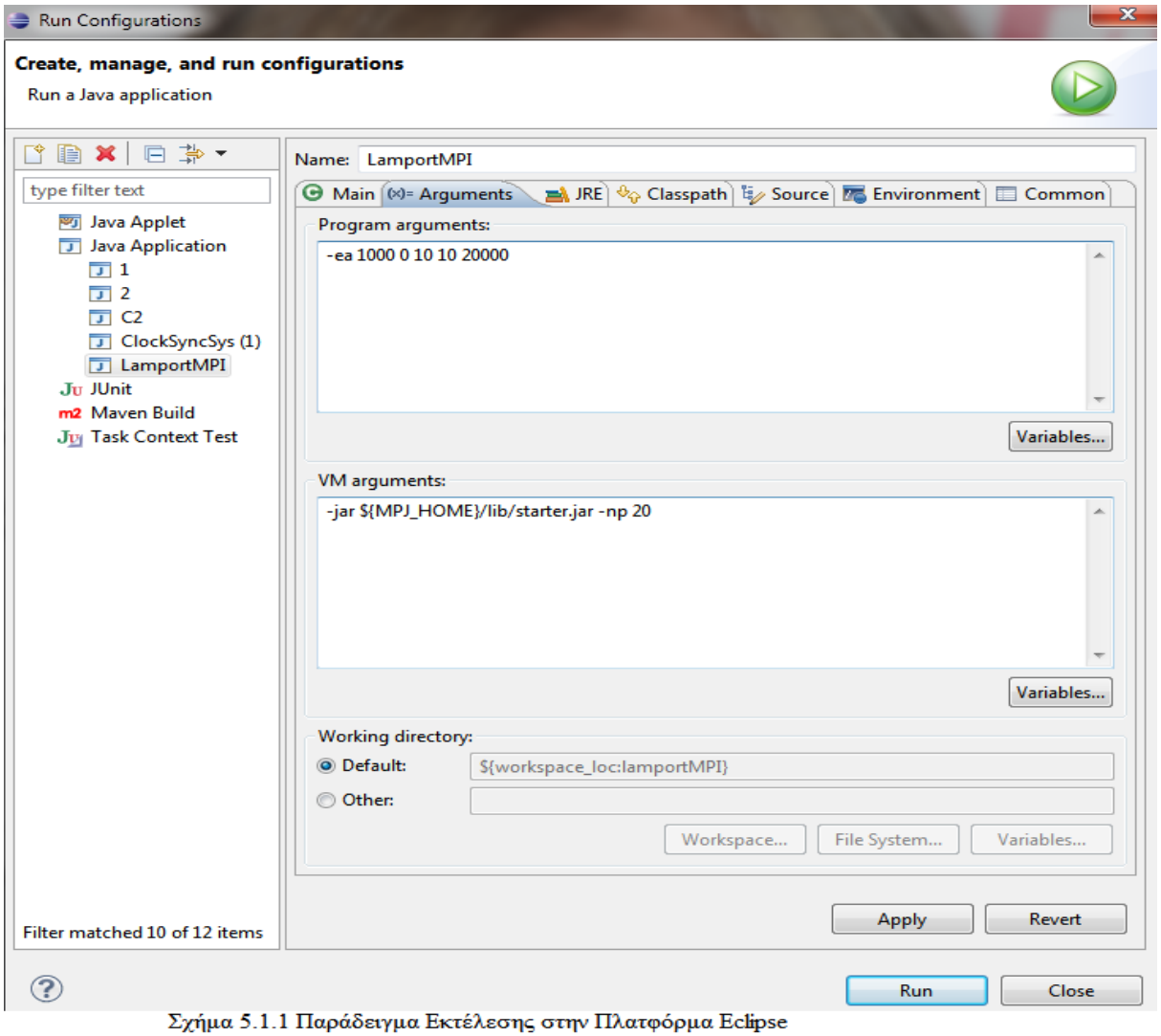
- Παράμετροι προγράμματος:

  -ea <program variables> (-dev niodev)

- Παράμετροι ιδεατής μηχανής:

  -jar ${MPJ_HOME}/lib/starter.jar -np <number_of_processes>

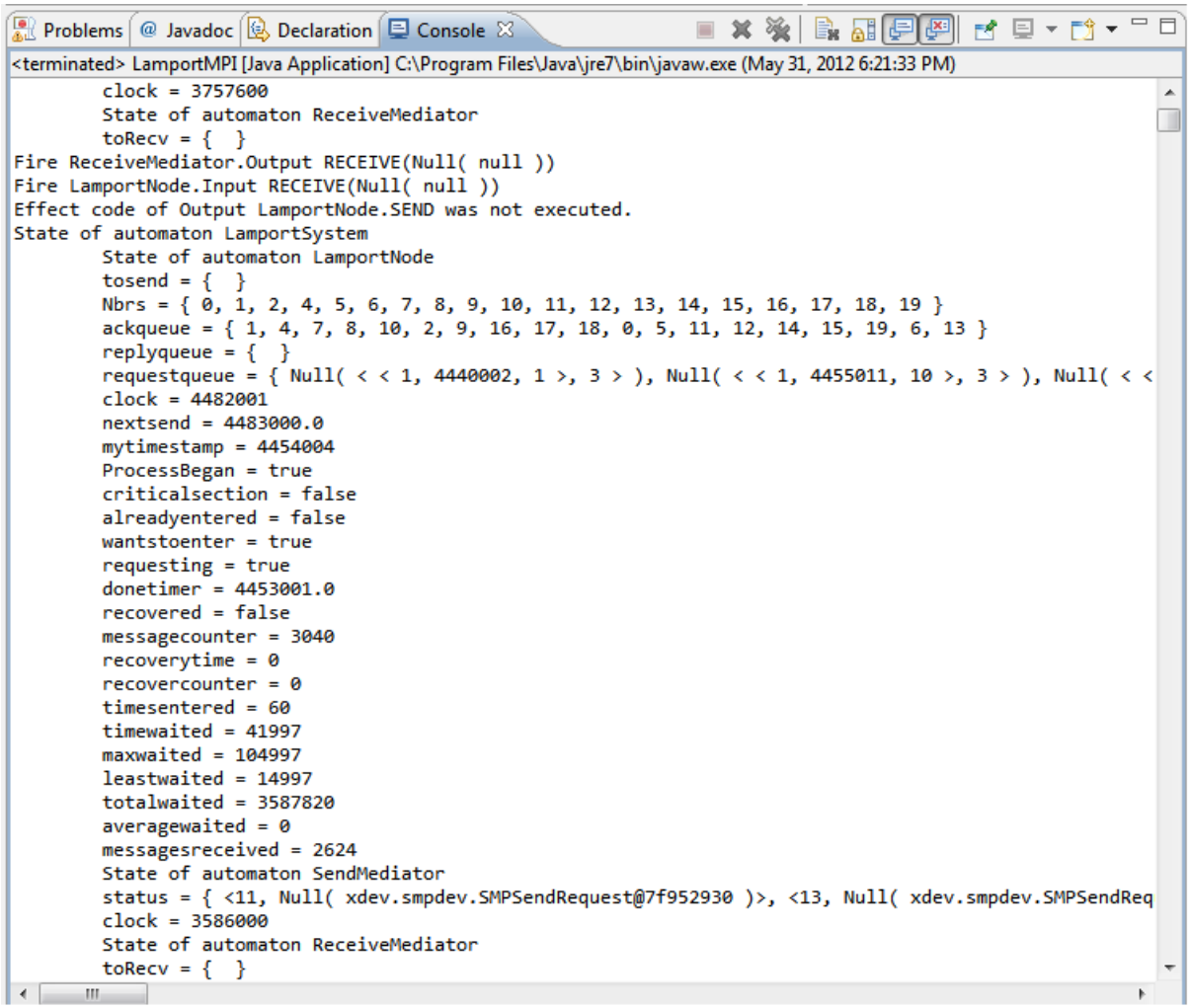
Όπου **<program variables>** μπαίνουν οι παράμετροι που δέχεται το πρόγραμμα Java, διαχωρισμένες με μονό διάστημα. Το τμήμα **(-dev niodev)** είναι προεραιτικό, μπαίνει μόνο σε περίπτωση που θα τρέξουμε το πρόγραμμα μας σε Cluster μηχανή. Τέλος, όπου **<number_of_processes>** μπαίνει ένας αριθμός, ο οποίος καθορίζει πόσα νήματα θα δημιουργηθούν. Στο σχήμα 5.1.1 φαίνεται ένα παράδειγμα ρύθμισης των παραμέτρων αυτών στην πλατφόρμα Eclipse για ένα από τα σενάρια που έτρεξα (συγκεκριμένα βλέπουμε τις ρυθμίσεις για το σενάριο Lamport με 20 διεργασίες χωρίς πιθανότητα σφάλματος).



Σχήμα 5.1.1 Παράδειγμα Εκτέλεσης στην Πλατφόρμα Eclipse

Η εκτέλεση των διεργασιών γίνεται παράλληλα. Τα αποτελέσματα φαίνονται στην κονσόλα της διαφάνειας του Eclipse, αλλά έχουμε την επιλογή να τα αποθηκεύουμε και σε αρχείο. Στο σχήμα 5.1.2 φαίνεται ένα στιγμιότυπο μιας διεργασίας Lamport κατά τη διάρκεια

εκτέλεσης ενός σεναρίου, όπως αυτό εκτυπώνεται στην κονσόλα. Συγκεκριμένα, παρουσιάζεται η κατάσταση της διεργασίας τρία σε μια τυχαία χρονική στιγμή, όπου και περιμένει έγκριση για να μπει στην κρίσιμη περιοχή.



Σχήμα 5.1.2 Στιγμιότυπο Εκτέλεσης Αλγόριθμου του Lamport με 20 κόμβους

5.2 Σενάρια Προσομοίωσης και Αποτελέσματα

Για τα σενάρια προσομοίωσης μου θέλω κυρίως να δω τον συνολικό αριθμό μηνυμάτων που στάλθηκαν κατά τη διάρκεια μιας εκτέλεσης, καθώς και τον μέσο χρόνο αναμονής της κάθε διεργασίας από την ώρα που κάνει την αίτηση να μπει στην κρίσιμη περιοχή μέχρι την ώρα που τα καταφέρνει. Επίσης, είναι ενδιαφέρον να δούμε κατά πόσο τα νούμερα αυτά επηρεάζονται από πιθανές καταρρεύσεις διεργασιών. Τέλος, είναι ενδιαφέρον να δούμε κατά πόσο ο αριθμός των διεργασιών που συμμετέχουν αποτελεί σημαντικό παράγοντα ή όχι. Για το λόγο αυτό, θα τρέξω σενάρια με περιπτώσεις κατάρρευσης (chancetodrop = 2%) και σενάρια χωρίς περιπτώσεις κατάρρευσης (chancetodrop = 0%) για τους δύο αλγόριθμους με 5,10 και 20 διεργασίες. Η τυχαία επιθυμία για κάθε διεργασία να μπει στην κρίσιμη περιοχή σε κάθε επανάληψη θα οριστεί στο 10% (chancetoenter = 10). Στα σενάρια με περιπτώσεις

αποτυχίας κόμβων έθεσα τον χρόνο αναμονής μέχρι να θεωρήσουμε μια διεργασία νεκρύ σε 10 κύκλους (timetotimeout=10). Αρχικά θα αναλύσω τον αλγόριθμο του Lamport, στην συνέχεια θα αναλύσω τον αλγόριθμο του Lelann και τέλος θα συγκρίνω τους δύο αυτούς αλγόριθμους.

5.2.1 Ανάλυση του Αλγόριθμου του Lamport

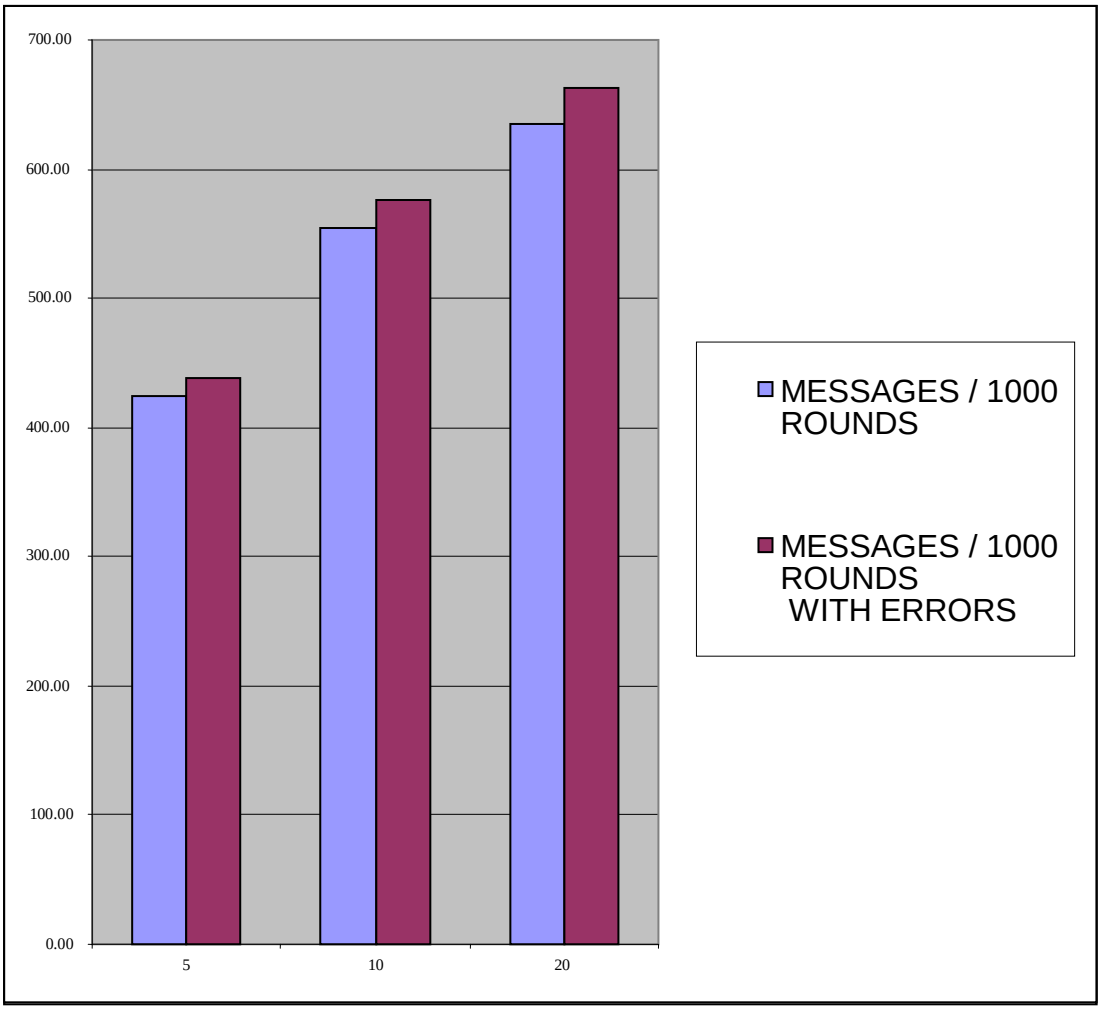
Στην ενότητα αυτή θα αναλύσω τα αποτελέσματα που πήρα τρέχοντας τα σενάρια του αλγόριθμου Lamport που προανέφερα. Τα σενάρια εκτελέστηκαν για 5000 επαναλήψεις (runs = 5000), με ρυθμό εξέλιξης της τροχιάς u=1000, πιθανότητα επιθυμίας εισόδου στην κρίσιμη περιοχή 10% (chancetoenter = 10). Στα σενάρια με σφάλματα, η κάθε διεργασία είχε 2% πιθανότητα σε κάθε επανάληψη να πάθει σφάλμα (chancetodrop = 2) και ο χρόνος αναμονής μέχρι να θεωρήσουμε μια γειτονική διαδικασία "νεκρή" ορίστηκε στους 10 κύκλους επανάληψης (timeoutwait = 10). Στον πίνακα που ακολουθεί παρουσιάζονται τα αποτελέσματα από τα σενάρια με 5, 10 και 20 διεργασίες αντίστοιχα. Οι μετρήσεις που πήρα ήταν ο μέσος αριθμός μηνυμάτων που στέλνει η κάθε διεργασία στις 1000 επαναλήψεις, καθώς και ο μέσος χρόνος αναμονής του κάθε κόμβου από την ώρα που εκφράζει την επιθυμία να εισέλθει στην κρίσιμη περιοχή μέχρι την ώρα που τελικά τα καταφέρνει. Ο χρόνος αυτός μετρήθηκε σε σχέση με το ρολόι και όχι με κύκλους.

LAMPORT PROCESSES	MESSAGES / 1000 ROUNDS	MESSAGES / 1000 ROUNDS WITH ERRORS	AVERAGE TIME WAITED	AVERAGE TIME WAITED WITH ERRORS
5	425.00	439.00	13330.33	13749.00
10	554.00	575.75	28408.09	40671.24
20	634.88	663.28	117703.58	175435.94

Στο σχήμα 5.2.1.1 παρουσιάζεται η γραφική παράσταση για τον μέσο αριθμό μηνυμάτων που στέλνει η κάθε διεργασία σε 1000 επαναλήψεις του αλγόριθμου. Παρατηρούμε ότι αυξάνοντας τον αριθμό των διεργασιών που συμμετέχουν στο σύστημα αυξάνεται γραμμικά και ο αριθμός των μηνυμάτων που αποστέλλονται ανά διεργασία. Αυτό είναι λογικό γιατί κάθε αίτηση και κάθε απάντηση πρέπει πλέον να αποστέλνεται σε περισσότερες διεργασίες. Επίσης, με την εισαγωγή πιθανότητας βλάβης, ο αριθμός των μηνυμάτων αυξάνεται ελαφρά, αφού σε κάθε περίπτωση βλάβης γίνεται κάποια ανταλλαγή μηνυμάτων για επαναφορά της διεργασίας σε λειτουργία.

Εδώ πρέπει να κάνουμε την παρατήρηση πως ο συνολικός αριθμός των μηνυμάτων που θα κυκλοφορεί στο δίκτυο αυξάνεται πλέον εκθετικά, μιας και η γραφική παράσταση δείχνει τον αριθμό μηνυμάτων ανά διεργασία, και όχι για τον συνολικό αριθμό μηνυμάτων που θα

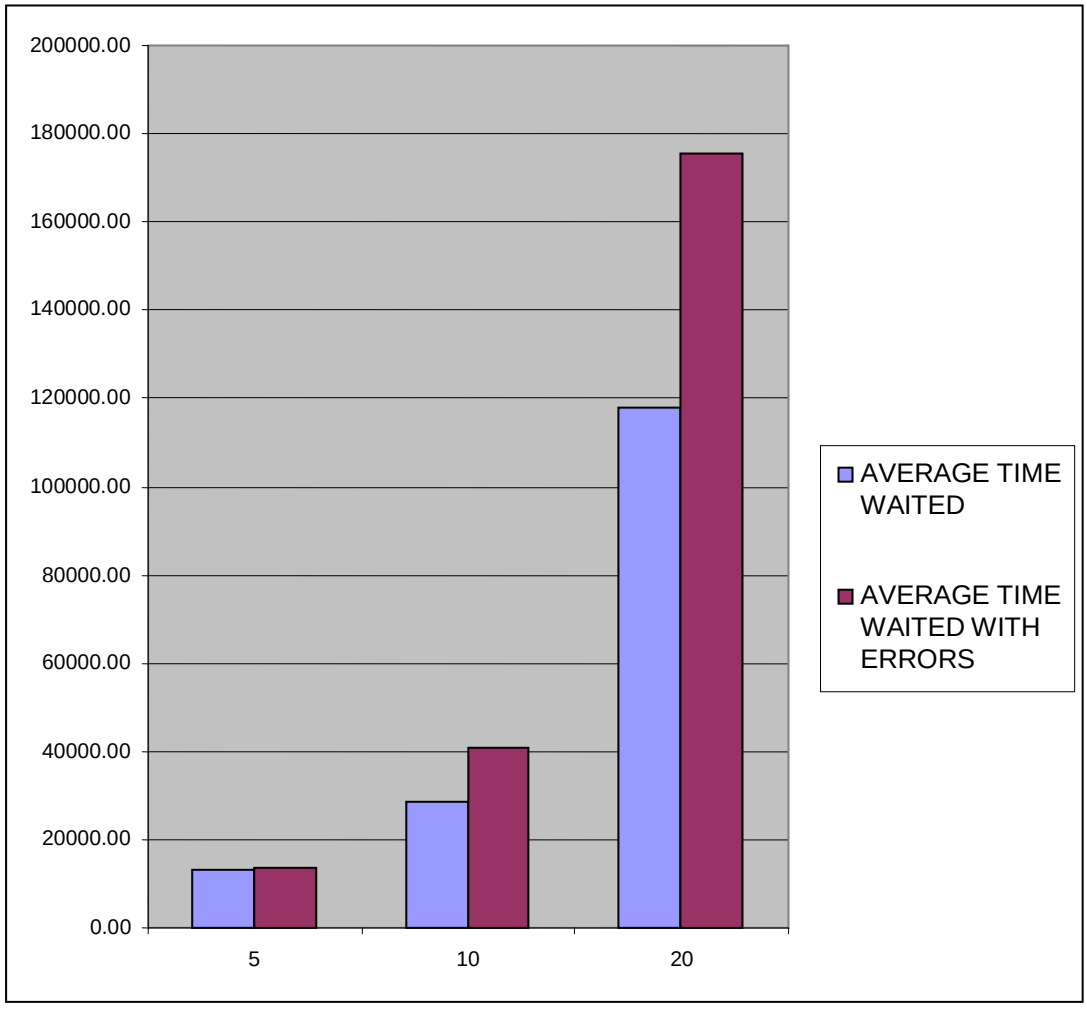
αποσταλούν στο δίκτυο. Αυτό ωστόσο δεν αποτελεί πρόβλημα, μιας και αναφερόμαστε σε πλήρως συνδεδεμένες τοπολογίες. Όσο αυξάνεται ο αριθμός των διεργασιών αυξάνεται και το μέγεθος του δικτύου (το πλήθος δηλαδή των γραμμών επικοινωνίας), κι έτσι δεν έχουμε υποχρησιμοποίηση του δικτύου.



Σχήμα 5.2.1.1 Γραφική Παράσταση για Αριθμό Μηνυμάτων Lamport

Στο σχήμα 5.2.1.2 βλέπουμε την γραφική παράσταση για τον μέσο όρο αναμονής της κάθε διεργασίας από την ώρα που εκφράζει την επιθυμία να μπει στην κρίσιμη περιοχή μέχρι την ώρα που τελικά τα καταφέρνει. Παρατηρούμε ότι ο μέσος χρόνος αναμονής αυξάνεται εκθετικά με την αύξηση του αριθμού των διεργασιών στο σύστημα. Αυτό είναι λογικό διότι η κάθε διεργασία θα πρέπει πλέον να πάρει έγκριση για είσοδο από περισσότερες διεργασίες, και επίσης αυξάνεται η πιθανότητα για δύο ή περισσότερες διεργασίες να αιτηθούν αίτηση σε κοντινά χρονικά πλαίσια, γεγονός που αυξάνει τον μέσο χρόνο αναμονής. Παρατηρούμε επίσης πως με την προσθήκη πιθανότητας σφαλμάτων αυξάνεται και πάλι ο μέσος χρόνος αναμονής. Αυτό συμβαίνει για το λόγο ότι εισάγεται πλέον και η έννοια του παράθυρου αναμονής στα μηνύματα, γεγονός που προσθέτει ακόμη περισσότερη καθυστέρηση στην ανταλλαγή μηνυμάτων. Για σενάρια με μεγαλύτερο αριθμό διεργασιών η διαφορά αυτή

αυξάνεται ακόμη περισσότερο λόγω του μεγαλύτερου αριθμού μηνυμάτων, στα οποία εισάγεται πλέον και η έννοια της αναμονής για έλεγχο σφάλματος.



Σχήμα 5.2.1.2 Γραφική Παράσταση για Μέσο Χρόνο Αναμονής Lamport

Τελειώνοντας με τον αλγόριθμο του Lamport, μπορούμε να εξάγουμε συνοπτικά τα εξής συμπεράσματα:

- Αποστέλλεται μεγάλος αριθμός μηνυμάτων για επικοινωνία μεταξύ διεργασιών. Ο αριθμός αυτός αυξάνεται εκθετικά με την αύξηση του αριθμού των διεργασιών που συμμετέχουν στο σύστημα.
- Σχετικά μικρός χρόνος αναμονής, η κάθε διεργασία μπορεί να αιτηθεί είσοδο στην κρίσιμη περιοχή αμέσως μόλις αποφασίσει ότι θέλει να εισέλθει.
- Γίνεται πλήρης χρήση του πλήρως συνδεδεμένου δικτύου, αφού απαιτείται από κάθε διεργασία να επικοινωνεί με όλες τις υπόλοιπες για κάθε αίτηση.
- Το επιπρόσθετο κόστος για το μηχανισμό εντοπισμού σφαλμάτων και αποκατάστασης διεργασιών αυξάνεται εκθετικά με την αύξηση του αριθμού των διεργασιών που συμμετέχουν στο σύστημα.

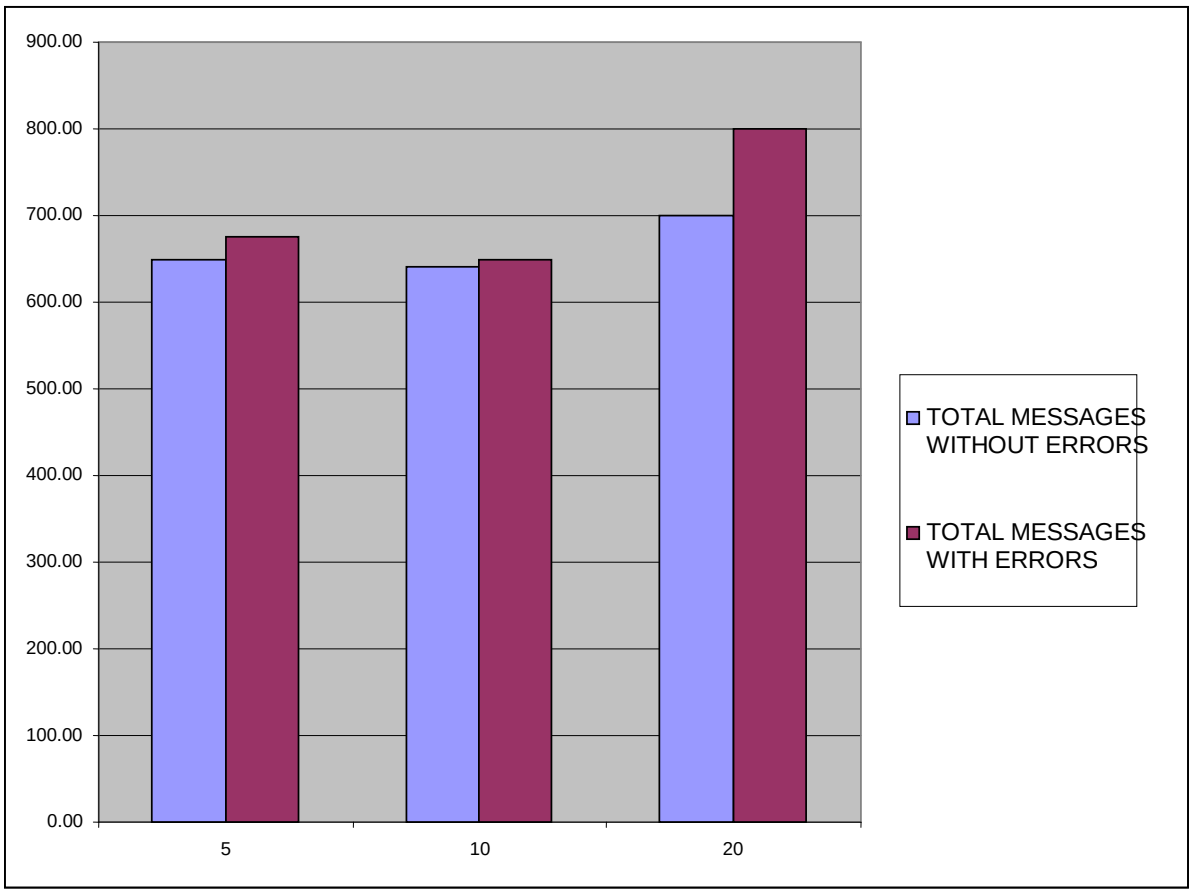
5.2.2 Ανάλυση του Αλγόριθμου του Lelann

Στην ενότητα αυτή θα αναλύσω τα αποτελέσματα που πήρα τρέχοντας τα σενάρια του αλγόριθμου Lelann που προανέφερα. Τα σενάρια εκτελέστηκαν για 5000 επαναλήψεις (runs = 5000), με ρυθμό εξέλιξης της τροχιάς u=1000, πιθανότητα επιθυμίας εισόδου στην κρίσιμη περιοχή 10% (chancetoenter = 10). Στα σενάρια με σφάλματα, η κάθε διεργασία είχε 2% πιθανότητα σε κάθε επανάληψη να πάθει σφάλμα (chancetodrop = 2) και ο χρόνος αναμονής μέχρι να θεωρήσουμε μια γειτονική διαδικασία "νεκρή" ορίστηκε στους 10 κύκλους επανάληψης (timeoutwait = 10). Στον πίνακα που ακολουθεί παρουσιάζονται τα αποτελέσματα από τα σενάρια με 5, 10 και 20 διεργασίες αντίστοιχα. Οι μετρήσεις που πήρα ήταν ο συνολικός αριθμός των μηνυμάτων που αποστέλλονται καθ' όλη τη διάρκεια της προσομοίωσης, καθώς και ο μέσος χρόνος αναμονής του κάθε κόμβου από την ώρα που εκφράζει την επιθυμία να εισέλθει στην κρίσιμη περιοχή μέχρι την ώρα που τελικά τα καταφέρνει. Ο χρόνος αυτός μετρήθηκε σε σχέση με το ρολόι και όχι με κύκλους.

LELANN PROCESSES	TOTAL MESSAGES SENT	TOTAL MESSAGES SENT WITH ERRORS	AVERAGE TIME WAITED	AVERAGE TIME WAITED WITH ERRORS
5	650.00	439.00	47480.00	53525.24
10	640.00	575.75	124500.00	149400.00
20	700.00	663.28	244000.00	322147.97

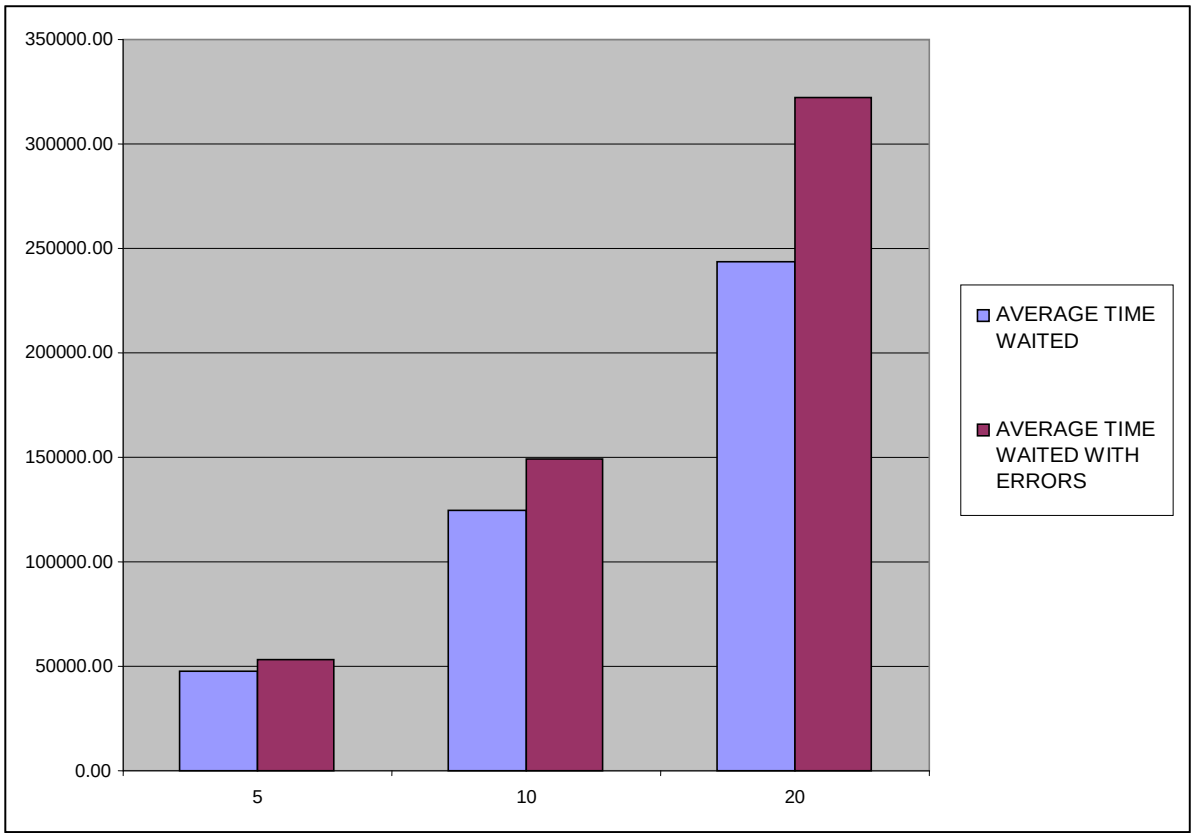
Στο σχήμα 5.2.2.1 παρουσιάζεται η γραφική παράσταση για τον αριθμό μηνυμάτων που αποστέλλονται σε 5000 επαναλήψεις του αλγόριθμου. Παρατηρούμε ότι αυξάνοντας τον αριθμό των διεργασιών που συμμετέχουν στο σύστημα ο αριθμός των μηνυμάτων που αποστέλλονται παραμένει σταθερός. Αυτό είναι λογικό γιατί σε κάθε γύρο αποστέλλονται το πολλύ ένα μήνυμα σκυτάλης και ένα μήνυμα αναγνώρισης. Επίσης, με την εισαγωγή πιθανότητας βλάβης, ο αριθμός των μηνυμάτων αυξάνεται γραμμικά, αφού κάθε κόμβος σε κάθε εκτέλεση έχει την πιθανότητα να πάθει βλάβη, στην οποία περίπτωση θα πρέπει να αποστείλει μήνυμα <recovery> μόλις επανέλθει. Το κόστος φαίνεται ελαφρώς μεγάλο γιατί ο αριθμός των μηνυμάτων είναι μικρός. Στην ουσία όμως το κόστος επικοινωνίας για επαναφορά των κόμβων είναι ευθέως ανάλογο του αριθμού των διεργασιών που συμμετέχουν στο σύστημα.

Εδώ πρέπει να κάνουμε την παρατήρηση ότι έχουμε υποχρησιμοποίηση του δικτύου, μιας και έχουμε ένα πλήρως συνδεδεμένο δίκτυο αλλά αποστέλλουμε μόνο 2-3 μηνύματα σε κάθε γύρο εκτέλεσης.



Σχήμα 5.2.2.1 Συνολικός Αριθμός Μηνυμάτων του Αλγόριθμου Lelann

Στο σχήμα 5.2.2.2 βλέπουμε την γραφική παράσταση για τον μέσο όρο αναμονής της κάθε διεργασίας από την ώρα που εκφράζει την επιθυμία να μπει στην κρίσιμη περιοχή μέχρι την ώρα που τελικά τα καταφέρνει. Παρατηρούμε ότι ο μέσος χρόνος αναμονής αυξάνεται εκθετικά με την αύξηση του αριθμού των διεργασιών στο σύστημα. Αυτό είναι λογικό διότι η κάθε διεργασία θα πρέπει πλέον να περιμένει περισσότερο για να λάβει τη σκυτάλη. Παρατηρούμε επίσης πως με την προσθήκη πιθανότητας σφαλμάτων αυξάνεται και πάλι ο μέσος χρόνος αναμονής. Αυτό συμβαίνει για το λόγο ότι εισάγεται πλέον και η έννοια του παράθυρου αναμονής στα μηνύματα, γεγονός που προσθέτει ακόμη περισσότερη καθυστέρηση στην ανταλλαγή μηνυμάτων. Για σενάρια με μεγαλύτερο αριθμό διεργασιών η διαφορά αυτή αυξάνεται ακόμη περισσότερο λόγω του μεγαλύτερου αριθμού μηνυμάτων, στα οποία εισάγεται πλέον και η έννοια της αναμονής για έλεγχο σφάλματος.



Σχήμα 5.2.2.2 Γραφική Παράσταση για Μέσο Χρόνο Αναμονής Lelann

Τελειώνοντας με τον αλγόριθμο του Lelann, μπορούμε να εξάγουμε συνοπτικά τα εξής συμπεράσματα:

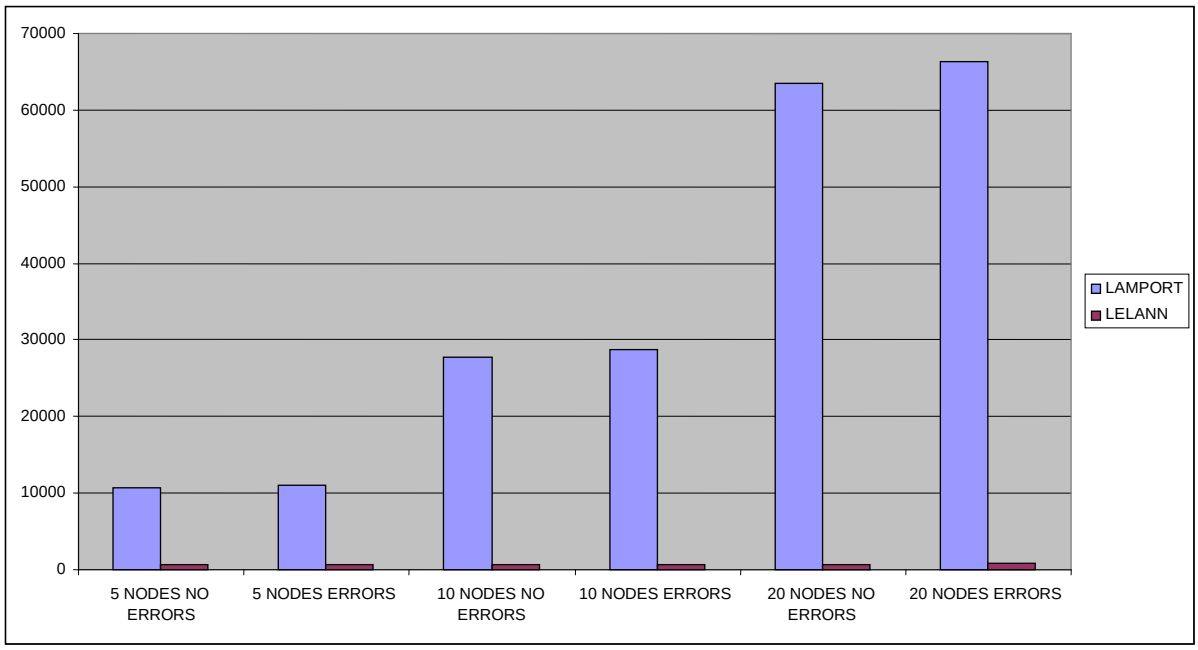
- Αποστέλλεται σταθερός (και πολύ μικρός) αριθμός μηνυμάτων για επικοινωνία μεταξύ διεργασιών. Ο αριθμός αυτός είναι ανεξάρτητος από τον αριθμό των διεργασιών που συμμετέχουν στο σύστημα.
- Σχετικά μεγάλος χρόνος αναμονής, η κάθε διεργασία πρέπει να περιμένει περιφορά της σκυτάλης για να εισέλθει στην κρίσιμη περιοχή.
- Δεν γίνεται πλήρης χρήση του πλήρως συνδεδεμένου δικτύου, αφού αποστέλλονται ελάχιστα μηνύματα.
- Το επιπρόσθετο κόστος για το μηχανισμό εντοπισμού σφαλμάτων και αποκατάστασης διεργασιών αυξάνεται γραμμικά με την αύξηση του αριθμού των διεργασιών που συμμετέχουν στο σύστημα.

5.2.3 Σύγκριση των Αλγορίθμων

Στην ενότητα αυτή θα συγκρίνω τις επιδόσεις των δύο αλγορίθμων ως προς τον συνολικό αριθμό των μηνυμάτων που παράγουν κατά τη διάρκεια μιας προσομοίωσης, καθώς και ως

προς τον μέσο χρόνο αναμονής της κάθε διεργασίας για είσοδο της στην κρίσιμη περιοχή. Οι μετρήσεις για τα μηνύματα φαίνονται στον πίνακα που ακολουθεί.

ΣΥΝΟΛΙΚΟΣ ΑΡΙΘΜΟΣ ΜΗΝΥΜΑΤΩΝ	LAMPORT	LELANN
5 NODES NO ERRORS	10625	650
5 NODES ERRORS	10975	675
10 NODES NO ERRORS	27700	640
10 NODES ERRORS	28787.5	650
20 NODES NO ERRORS	63488	700
20 NODES ERRORS	66328	800



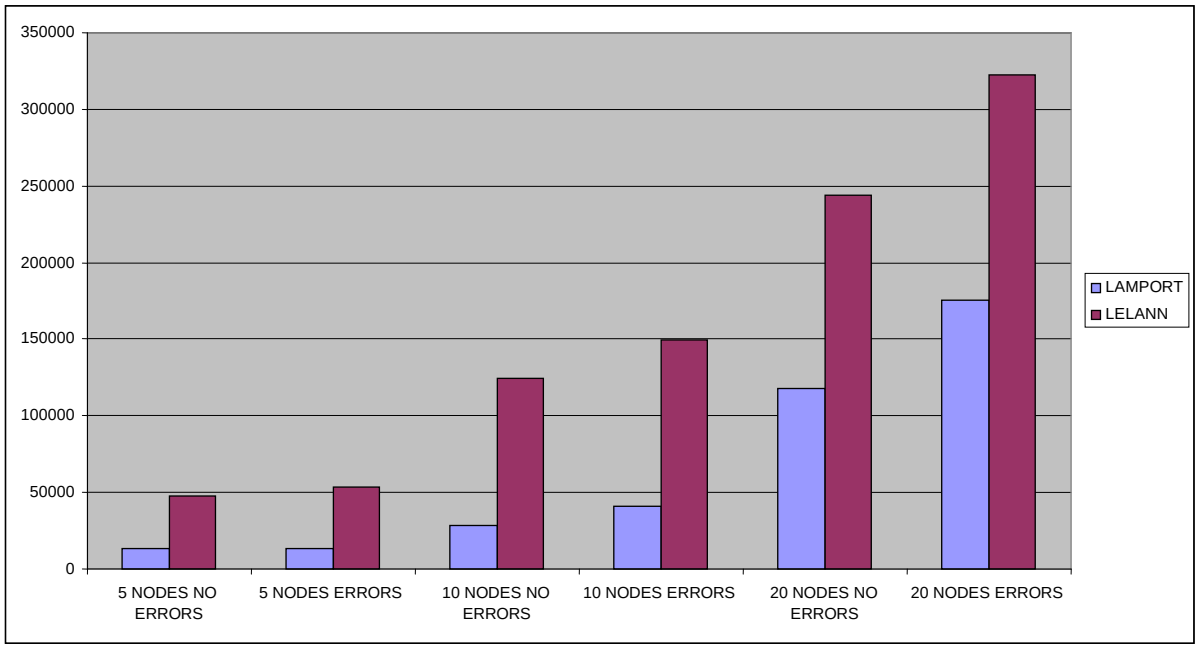
Σχήμα 5.2.3.1 Συνολικός Αριθμός Μηνυμάτων για τους δύο Αλγόριθμους

Στο σχήμα 5.2.3.1 φαίνεται ο αριθμός των μηνυμάτων που στέλλονται κατά τη διάρκεια μιας προσομοίωσης (5000 επαναλήψεις) για τους αλγόριθμους που εξετάζονται. Παρατηρούμε ότι ο αλγόριθμος του Lelann αποστέλλει πολύ μικρότερο αριθμό μηνυμάτων από ότι τον αντίστοιχο του Lamport. Αυτό έγκειται στο γεγονός ότι σε κάθε επανάληψη του αλγόριθμου του Lamport η κάθε διεργασία μπορεί να στείλει μέχρι n-1 μηνύματα αιτήσεων (δηλαδή μπορεί να έχουμε συνολικά μέχρι n * (n-1) μηνύματα) ενώ στον αλγόριθμο του Lelann μπορούμε να έχουμε μόνο δύο μηνύματα σε κάθε επανάληψη (το μήνυμα σκυτάλη και ένα μήνυμα επιβεβαίωσης). Για περιπτώσεις σφαλμάτων υπάρχει ένα μικρό επιπρόσθετο κόστος για τον κάθε κόμβο που συμμετέχει στον κάθε αλγόριθμο. Το κόστος του Lamport είναι λίγο πιο ψηλό γιατί το μήνυμα <recovery> πρέπει να σταλεί σε όλες τις υπόλοιπες διεργασίες που συμμετέχουν, ενώ στον αλγόριθμο του Lelann το μήνυμα αυτό αποστέλλεται μόνο σε μια διεργασία κάθε φορά.

Στη συνέχεια θα αναλύσω τον μέσο όρο αναμονής του κάθε κόμβου από την ώρα που αιτείται πρόσβαση στην κρίσιμη περιοχή μέχρι την ώρα που αποκτά την πρόσβαση αυτή. Οι μετρήσεις που πήρα από τα σενάρια που έτρεξα φαίνονται στον πίνακα που ακολουθεί.

ΜΕΣΟΣ ΧΡΟΝΟΣ ΑΝΑΜΟΝΗΣ	**LAMPORT**	**LELANN**
5 NODES NO ERRORS	13330.33	47480
5 NODES ERRORS	13749	53525.24
10 NODES NO ERRORS	28408.09	124500
10 NODES ERRORS	40671.24	149400
20 NODES NO ERRORS	117703.6	244000
20 NODES ERRORS	175435.9	322147.97

Στο σχήμα 5.2.3.2 φαίνεται η γραφική παράσταση για το μέσο χρόνο αναμονής των διεργασιών στους δύο αλγόριθμους. Παρατηρούμε πως οι χρόνοι του αλγόριθμου Lelann είναι κατά πολύ ψηλότεροι από αυτούς του Lamport. Αυτό συμβαίνει γιατί στον αλγόριθμο Lamport η αίτηση γίνεται κατευθείαν μόλις η διεργασία επιθυμεί να μπει στην κρίσιμη περιοχή, ενώ στον αλγόριθμο του Lelann η "αίτηση" γίνεται στην ουσία αφού η διεργασία λάβει τη σκυτάλη. Το κόστος ωστόσο του αλγόριθμου του Lamport για να παρέχει ψηλότερες ταχύτητες πρόσβασης είναι ο τεράστιος αριθμός μηνυμάτων που πρέπει να αποστέλλονται στο δίκτυο.



Σχήμα 5.2.3.2 Μέσος Χρόνος Αναμονής για τους 2 Αλγόριθμους

Κεφάλαιο 6

Συμπεράσματα

6.1 Γενικά Συμπεράσματα

Αναλύοντας τα αποτελέσματα που βγήκαν από τα σενάρια προσομοίωσης (όπως αυτά περιγράφονται στο κεφάλαιο 5) για τους δύο αλγόριθμους, μπορούμε να εξάγουμε κάποια συμπεράσματα για το πότε είναι κατάλληλη η χρήση του κάθε αλγόριθμου. Σε περιπτώσεις όπου κύριο μας μέλημα είναι να αποφύγουμε την ανταλλαγή πολλών μηνυμάτων προτείνεται ο αλγόριθμος του Lelann λόγω του σταθερού αριθμού μηνυμάτων ανεξαρτήτως αριθμού διεργασιών. Ο αριθμός των μηνυμάτων που ανταλλάσσονται με χρήση του αλγόριθμου του Lelann είναι πολύ πιο μικρός από τον αντίστοιχο του αλγόριθμου Lamport. Ωστόσο, θα πρέπει να αποδεκτούμε και το αντίστοιχο αντίτιμο, το γεγονός δηλαδή ότι οι διεργασίες θα χρειάζονται περισσότερο χρόνο να μπουν στην κρίσιμη περιοχή. Αν μας ενδιαφέρει κυρίως η ταχύτητα εξυπηρέτησης των διεργασιών από την κρίσιμη περιοχή τότε συστήνεται χρήση του αλγόριθμου του Lamport. Τέλος, παρατηρούμε ότι και στους δύο αλγόριθμους υπάρχει ένα επιπρόσθετο κόστος για να υλοποιήσουμε τη λειτουργία αποκατάστασης κόμβων μετά από βλάβη, όμως το κόστος αυτό είναι απαραίτητο για την ορθή λειτουργία και των δύο αλγορίθμων, ειδικά σε περιπτώσεις που θέλουμε να τους εφαρμόσουμε σε δίκτυα με κόμβους οι οποίοι βρίσκονται διεσπαρμένοι σε μεγάλη γεωγραφική έκταση.

6.2 Προβλήματα που αντιμετωπίσθηκαν

Κατά τη διάρκεια της εκπόνησης της διπλωματικής αυτής εργασίας αντιμετώπισα συγκεκριμένα προβλήματα, τα οποία και αναφέρω πιο κάτω:

1. Σε αρχικά στάδια της διπλωματικής εργασίας ασχολήθηκα με τον προσομοιωτή για να αναπτύξω τους αλγόριθμους μου. Ο προσομοιωτής δεν βρίσκεται ακόμη σε ολοκληρωμένο στάδιο, και πολλές εντολές δεν λειτουργούσαν σωστά. Περισσότερα αναφέρονται στο [4].
2. Η εντολή choose...where δεν λειτουργεί όπως πρέπει. Παρόλο που όριζα περιορισμούς, η εντολή εξακολουθούσε να μου επιστρέφει αυθαίρετους αριθμούς.
3. Αφού είχα υλοποιήσει τους αλγόριθμους μου για χρήση με τον προσομοιωτή, προσπάθησα να τους μεταφέρω σε java με τη χρήση του εργαλείου tempo2java, όμως οι προδιαγραφές του εργαλείου είναι ακόμη περιορισμένες, κι έτσι έπρεπε ολόκληρος ο κώδικας να ξαναγραφτεί. Αυτό γιατί ο προσομοιωτής "κοιτάζει" την εκτέλεση της προσομοίωσης από διαφορετική οπτική γωνία. Για παράδειγμα, για να τρέξω τον αλγόριθμο του Lamport με 3 κόμβους στον προσομοιωτή θα έπρεπε να δημιουργήσω ένα σύνθετο αυτόματο με 3 αυτόματα τύπου LamportNode και 6 αυτόματα τύπου Channel, όπως αυτά ορίζονται σε έτοιμο πακέτο που περιλαμβάνεται στο εργαλείο Tempo, ενώ στην περίπτωση του μεταφραστή θα μεταφράσω τον κώδικα εκτέλεσης ενός μόνο κόμβου. Επίσης, είχα χρησιμοποιήσει διαφορετικά κανάλια, τα οποία δεν μπορούσα να χρησιμοποιήσω με τον μεταφραστή java.
4. Η αποσφαλμάτωση των προδιαγραφών με δυσκόλεψε πάρα πολύ λόγω του τεράστιου όγκου δεδομένων που τυπώνονται στην οθόνη. Για παράδειγμα, για ένα σενάριο 250 επαναλήψεων ο κάθε κόμβος δημιουργούσε ένα log file μεγέθους 24 Megabyte, το οποίο στη συνέχεια έπρεπε να μελετήσω για να εντοπίσω το σφάλμα. Γενικά, για κάθε εκτέλεση μιας ενέργειας μεταβάσεων (στον προγραμματιστή ενεργειών και τροχιών με την λέξη-κλειδί "fire") τυπώνεται στην οθόνη ολόκληρη η κατάσταση του αυτόματου (δηλαδή οι τιμές όλων των μεταβλητών και τα περιεχόμενα όλων των ουρών). Αν και χρήσιμο για αποσφαλμάτωση, θα ήταν καλό να μας παρέχεται η επιλογή να περιορίσουμε ή ακόμη και να απενεργοποιήσουμε τελείως όλες αυτές τις εκτυπώσεις στην οθόνη.

6.3 Μελλοντική εργασία

Θα ήταν χρήσιμο να δοκιμαστούν οι προδιαγραφές των αλγορίθμων μου σε μια ανοικτή πλατφόρμα όπως είναι το Planet-lab. Το Planet-lab είναι μια πλατφόρμα στην οποία ανήκουν διάφοροι φυσικοί πόροι (κόμβοι) διεσπαρμένοι κατά μήκος της υφηλίου. Η πλατφόρμα Planet-lab μας επιτρέπει να τρέξουμε κώδικες σε διεσπαρμένες γεωγραφικά μονάδες με πραγματική καθυστέρηση ανταλλαγής μηνυμάτων λόγω της φυσικής απόστασης. Επίσης, το

εργαλείο Tempo θα ήταν χρήσιμο να αναπτυχθεί περαιτέρω. Για παράδειγμα, θα μπορούσε να υλοποιηθεί κάποιο γραφικό περιβάλλον προσομοίωσης το οποίο να μας επιτρέπει να δούμε οπτικά μια προσομοίωση εν ώρα λειτουργίας, κάτι το οποίο θα μας διευκολύνει τρομερά στο θέμα της αποσφαλμάτωσης. Τέλος, ο μεταφραστής δεν βρίσκεται ακόμη σε ολοκληρωμένο στάδιο, υπάρχει ακόμη δουλειά που πρέπει να γίνει για να ολοκληρωθεί η εργασία αυτή.

Βιβλιογραφία

[1] Dilsun K. Kaynar, Nanacy Lynch, Roberto Segala, Frits Vaandrager. "The Theory of Timed I/O Automata." Morgan & Claypool Publishers, 2010.

[2] Ι. Κ. Κάβουρας, Ι. Ζ. Μηλής, Γ. Β. Ξυλωμένος και Α. Α. Ρουκουνάκη. Κατανεμημένα συστήματα με Java, Συστήματα Υπολογιστών Τόμος ΙΙΙ. Έκδοση 2η, Εκδόσεις Κλειδάριθμος. Αθήνα 2005.

[3] Nancy A. Lynch, Stephen J. Garland, Dilsun Kaynar, Laurent Michel, Alex Shvartsman. The Tempo Language User Guide and Reference Manual, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, October 24 2011.

[4] Στέφανος Ατριαν Ματεϊ. Προδιαγραφή και προσομοίωση ενός αλγόριθμου δρομολόγησης σε Ad-Hoc δίκτυα με τη χρήση του εργαλείου Tempo - TIOA. Ατομική Διπλωματική Εργασία, Πανεπιστήμιο Κύπρου, Τμήμα Πληροφορικής, Ιούνιος 2008.

[5] Peter M. Musial. Using Timed Input/Output Automata for Implementing Distributed Systems, Computer Science and Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 2011.

[6] Χρίστος Πλούταρχου. Enhancing the TEMPO compiler to support Java-Sockets / TCP-Based Communication. Διατριβή Master, Πανεπιστήμιο Κύπρου, Τμήμα Πληροφορικής, Ιούνιος 2011.

[7] Tempo - toolset homepage. Available at http://www.veromodo.com.

[8] MPJ Express homepage. Available at http://mpj-express.org.

Παράρτημα Α

Α.1 Κώδικας για το MPI Receiver Mediator (ReceiveMediator.tioa)

```
automaton ReceiveMediator
signature
    output RECEIVE(m:Null[mpi_message])
    input  probe(s:Nat)
states
    toRecv:Seq[mpi_message] := {};
transitions
    output RECEIVE(m) where len(toRecv) = 0
    pre
        m = nil();

    output RECEIVE(m) where len(toRecv) ~= 0
    pre
        m = embed(head(toRecv));
    eff
        toRecv := tail(toRecv);

    input probe(s)
    locals
      status:Null[mpi_status] := nil();
    eff
      status := MPI_Iprobe(s);
       if (status ~= nil()) then
          if (MPI_Test(val(status))) then
             toRecv := toRecv |- MPI_Irecv(val(status),s);
          fi
       fi
```

A.2 Κώδικας για το MPI Send Mediator (SendMediator.tioa)

```
automaton SendMediator
 signature
       input SEND(m:Null[mpi_message])
 states
       status:Array[Nat, Null[mpi_request]] := constant(nil());
       clock:AugmentedReal := 0;
 transitions
       input SEND(m)
       eff
              if (m ~= nil()) then
                 status[val(m).destination] := MPI_Isend(val(m),
val(m).destination);
              fi

  trajectories
    trajdef DELAY
      evolve d(clock) = 1;
```

Α.3 Κώδικας για το λεξιλόγιο TCP (TCPVocabs.tioa)

```
%%% .: algorithm vocabs :.
vocabulary alg_specific_voc
  types %Index : Enumeration [p1, p2, p3, p4],
              Data  : Tuple[type: Nat, timestamp: DiscreteReal , sender: Nat]
end

%%% .:MPI Channel vocabs:.
vocabulary mpi_status_voc
      types mpi_status
      operators
             MPI_Iprobe : Nat -> Null[mpi_status],
             MPI_Test : mpi_status -> Bool
end

vocabulary mpi_message_voc
    imports alg_specific_voc, mpi_status_voc
      types mpi_message : Tuple[data:Data, destination:Nat]
      operators
          MPI_Irecv : mpi_status, Nat -> mpi_message
end

vocabulary mpi_request_voc
    imports mpi_message_voc
      types mpi_request
      operators
          MPI_Isend : mpi_message, Nat -> Null[mpi_request],
          MPI_Barrier : -> Bool
end

vocabulary mpi_voc
   operators
      MPI_Rank : -> Nat,
      MPI_Size : -> Nat
end
```

Παράρτημα Β

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%% .:Lamport Node
%
%%
%% Describes an internal Node as implemented for Lamport Algorithm
%% in distributed systems. It is fault tolerant.
%
%
%%% .: ALGORITHM AUTOMATON :.
%%% u --> a frequency of advancing the trajectory
%%% chancetodrop --> A percentage chance for this node to drop in each iteration.
for example 1 -> 1%.
%%% chancetoenter --> A percentage chance for this node to want to enter the
critical region
%%% in each iteration. for example 1 -> 1%.
%%% rank --> The rank of this node.
%
%
%the msg field will be used as a means of defining the type of message sent.
% 1 - Request message
% 2 - Acknowledge message
% 3 - Release message
% 4 - Recovery message
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

automaton LamportNode( u, chancetodrop, chancetoenter,timeoutwait: DiscreteReal,
rank, size: Nat)
  signature
    output SEND( m:Null[mpi_message] )
    input  RECEIVE( m:Null[mpi_message] )
    internal prepmessages ( type: Nat), init, entercritical, checkfail,
checkenter, checktimeout,timer
  states
    tosend:                       Seq[Null[mpi_message]] := { }; %the queue of
messages to be sent
    Nbrs:                  Seq[Nat] := { }; % Includes all the nodes of the system
    ackqueue:              Seq[Nat] := { }; % Stores all the nodes that sent the
ack signal
    replyqueue:                   Seq[Nat] := { }; % Stores all the nodes that
await a reply from this node
    requestqueue:          Seq[Null[mpi_message]] := { }; % Holds all the requests
of entry for all the nodes
    clock:                        DiscreteReal := 0; % The physical clock of the
automaton
    nextsend:            DiscreteReal:=0; % Time to stop the trajectory
    mytimestamp:           DiscreteReal := 0; % The timestamp
    ProcessBegan:          Bool:=true; % Indicates a working process
    criticalsection:       Bool:=false; % Indicates entry into the critical region
    alreadyentered:        Bool:=false; % Used to avoid duplicated entry into the
critical region
    wantstoenter:          Bool:=false; % Indicates the wish to enter the critical
region
    requesting:                   Bool:=false; % Indicates that this process has
sent the request signals
    donetimer:                    DiscreteReal:=0; % A timer to tell when the
process is done with the critical region. Decided randomly.
```

```
    recovered:                  Bool:=false; % Indicates that the process has
recovered from a crash
    messagecounter:       Int:=0; % A counter to count how many messages this
node has sent
    recoverytime:         DiscreteReal:=0; % The time it will take to recover
after each failure. Decided randomly.
    recovercounter:       Int:=0; %counts how many times this process has
recovered from crash failures.
    timesentered:         Int:=0; %counts how many times this process has entered
the critical section.
    timewaited:                 DiscreteReal:=0; % Time waited from request to
entering for a single entrance into the critical section.
    maxwaited:                  DiscreteReal:=0; % Max time waited to enter.
    leastwaited:          DiscreteReal:=1000000; % Least time waited to enter.
    totalwaited:          DiscreteReal:=0; % Total time spent waiting to enter.
    messagesreceived:     Int:=0;


    initially u > 0;


  transitions

    output SEND(m)
      pre
         tosend ~= { };
         m = head(tosend);
         ProcessBegan;
      eff
         tosend := tail(tosend);
         messagecounter:=messagecounter+1;


    %Data  : Tuple[type: Nat, timestamp: DiscreteReal , sender: Nat]
    input RECEIVE(m)
      eff
        if m ~= nil() then
             messagesreceived:=messagesreceived+1;
             %Receive the request signal
                   %also adds the sender in the reply queue in order to send him
the ack signal
             if(val(m).data.type=1) then
                   requestqueue := requestqueue |- m;
                         replyqueue := replyqueue |- val(m).data.sender;
                   fi;
             %Receive the ack signal
             if (val(m).data.type=2) then
                   ackqueue := ackqueue |- val(m).data.sender;
             fi;
             %Receive the release signal from the process that currently was in
the critical region
                   %this should delete the request of that process from the
request queue
             if (val(m).data.type=3) then
                         for n : Nat where n < len(requestqueue) do
                               if
val(requestqueue[n]).data.sender=val(m).data.sender then
                                     requestqueue:= eject (requestqueue,n);
                               fi;
                         od;
                   fi;
                   %Receive the recovery signal
             if (val(m).data.type=4) then
                         for n : Nat where n < len(Nbrs) do
```

```
                                        if Nbrs[n]=val(m).data.sender then
                                              Nbrs:=eject (Nbrs,n);
                                        fi
                                    od;
                Nbrs := Nbrs |- val(m).data.sender;
                fi;
            fi


        internal prepmessages(type)
          pre len(tosend) = 0 /\ ProcessBegan;
          eff
                        if(type=1 /\ wantstoenter /\ ~requesting) then   %request
signal
                        for n : Nat where n < len(Nbrs) do
                        tosend := tosend |-
embed([[1,mytimestamp,MPI_Rank()],Nbrs[n]]);
                        od
                        requesting:=true;
                fi;
                if(type=2 ) then   %ack signal
                        for n : Nat where n < len(replyqueue) do
                        tosend := tosend |-
embed([[2,mytimestamp,MPI_Rank()],replyqueue[n]]);
                        od
                        replyqueue:={};
                fi;
                if(type=3 /\ criticalsection /\ clock >= donetimer) then   %release
signal
                        criticalsection:=false; %exit the critical section if you are
in it
                        alreadyentered:=false;
                        requesting:=false;
                        ackqueue:={};
                        for n : Nat where n < len(Nbrs) do
                        tosend := tosend |-
embed([[3,mytimestamp,MPI_Rank()],Nbrs[n]]);
                        od
                fi;
                if(type=4 /\ recovered) then   %recovery signal
                        recovered:=false;
                        for n : Nat where n < len(Nbrs) do
                        tosend := tosend |-
embed([[4,mytimestamp,MPI_Rank()],Nbrs[n]]);
                        od
                fi;


        internal init
        locals
           temp:Nat:=0;
           pre true;
          eff
             for n : Nat where n < MPI_Size() do
                if ~(n=MPI_Rank()) then
                        Nbrs:= Nbrs |- n;
                        Nbrs:= set(Nbrs,temp,n);
                        temp:=temp +1;
                 fi;

             od

         internal timer
           pre true;
```

```
      eff
         nextsend := nextsend + u;

    %enters the critical section only if the reply list is full(it got a reply
from each process)
      %and its own request has the smallest timestamp
      internal entercritical
      locals
       i:Int:=0;
       found:Bool:=false;
       proceed:Bool:=true;
      pre(ProcessBegan  /\ wantstoenter /\ ~criticalsection);
      eff

      for k : Nat where k < len(Nbrs) do
            found:=false;
            for j : Nat where j < len(ackqueue) do
                  if Nbrs[k]= ackqueue[j] then found:=true; fi;
            od
            if ~found then proceed:=false; fi;
      od

      if proceed then
            criticalsection:=true;
            wantstoenter:=false;
            for n : Nat where n < len(requestqueue) do
                  if val(requestqueue[n]).data.sender~=rank /\
val(requestqueue[n]).data.timestamp < mytimestamp  then
                        criticalsection:=false;
                        wantstoenter:=true;
                  fi;
            od;
            if criticalsection /\ ~alreadyentered then
                  %at this point we can assume that the process has successfully
entered the critical region
                  requesting:=false;
                  timesentered:=timesentered +1;
                  timewaited:=clock - mytimestamp;
                  if timewaited < leastwaited then leastwaited:=timewaited;fi;
                  if timewaited > maxwaited then maxwaited:=timewaited;fi;
            i:=choose i;
            i:= abs(i);
            i:=mod(i,10)+1;
                  donetimer:=clock+i*u;
                  alreadyentered:=true;
                  totalwaited:=totalwaited+timewaited;
            fi;
      fi;

      %randomly checks if the process will fail
      %the chance is given by the parameter chancetofail
      %also decides a random recovery time between 10 and 40 rounds
      %when the recovery time is there, the process will be enabled again
      internal checkfail
      locals
        i:Int:=0;
        j:Int:=0;
       pre ~criticalsection /\ chancetodrop>0;
      eff
        if(ProcessBegan) then
            i:=choose i;
            i:= abs(i);
            i:=mod(i,100);
            if i<chancetodrop then
```

```
                    ProcessBegan:=false;
                    j:= choose j;
                    j:=abs(j);
                    j:=mod(j,10)+5;
                    recoverytime:=clock + j*u;
              fi
        else
              if clock=recoverytime then
                    ProcessBegan:=true;
                    recovered:=true;
                    recovercounter:=recovercounter+1;
              fi
        fi

      %randomly checks if the process will want to enter the critical region
      %the chance is given by the parameter chancetoenter
      internal checkenter
      locals
        i:Int:=0;
       pre ProcessBegan /\ ~wantstoenter /\ ~criticalsection;
      eff
        i:=choose i;
        i:= abs(i);
        i:=mod(i,100);
        if i<chancetoenter then
              wantstoenter:=true;
              mytimestamp := clock+ MPI_Rank();
        fi


      %checks if some nodes timed out
      %compares the ackqueue with all the neighbors
      %is triggered 10 rounds after the process has requested entry
      internal checktimeout
      locals
       found:Bool:=false;
        pre ProcessBegan /\ requesting /\ clock >= (mytimestamp + timeoutwait *
u);
        eff
              for i : Nat where i < len(Nbrs) do
              found:=false;
              for j : Nat where j < len(ackqueue) do
                    if Nbrs[i] = ackqueue[j] then found:=true; fi;
                    od
              if ~found then Nbrs:=eject (Nbrs,i); fi;
              od

  trajectories
    trajdef T
       stop when clock>nextsend;
        evolve d(clock) = 1;
```

B.2 Κώδικας για ένα ολοκληρωμένο κόμβο Lamport (LamportSystem.tioa)

```
%%% .: VOCABS :.
include "MPIVocabs.tioa"

imports mpi_message_voc, mpi_request_voc, mpi_status_voc, mpi_voc,
alg_specific_voc

%%% .: MPI AUTOMATA :.
include "ReceiveMediator.tioa", "SendMediator.tioa"

%%% .: ALG AUTOMATON :.
include "LamportNode.tioa"

% -- eclipse running instructions
% program arguments:-ea 100 2 5 20 200
% for cluster we also need to add this parameter: -dev niodev
% VM arguments: -jar ${MPJ_HOME}/lib/starter.jar -np 5

automaton LamportSystem(u,chancetodrop,chancetoenter,timeoutwait: DiscreteReal,
runs:Nat)
  components
    LN : LamportNode(u,chancetodrop,chancetoenter,timeoutwait,MPI_Rank(),
MPI_Size());
    SM : SendMediator;
       RM : ReceiveMediator;

schedule
states
   m : Null[mpi_message] := nil();
   request   :Nat:=1;
   ack       :Nat:=2;
   release   :Nat:=3;
   recovery  :Nat:=4;

do

   fire internal LN.init;
   % -- From here starts the main loop of the execution
   for i : Nat where i < runs do
      follow LN.T duration \infty;

        % -- Fix the timer of this run
      fire internal LN.timer;

        % -- Check if the process will randomly fail
      fire internal LN.checkfail;

      % -- Check if the process randomly wants to enter the critical region
      fire internal LN.checkenter;

      % -- Check if some nodes timed out
      fire internal LN.checktimeout;


       % -- Prepare the request signals
      fire internal LN.prepmessages(request);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
```

```
    od
    follow SM.DELAY duration (MPI_Size() * 10);

    % -- Receive
    for j :Nat where j < MPI_Size() do
       m := nil();
       fire input  RM.probe( j );
          fire output RM.RECEIVE( m );
       od

     % -- Prepare the ack signals
    fire internal LN.prepmessages(ack);

     % -- Send
    for j :Nat where j < MPI_Size() do
       fire output LN.SEND(m);
    od
    follow SM.DELAY duration (MPI_Size() * 10);

    % -- Receive
    for j :Nat where j < MPI_Size() do
       m := nil();
       fire input  RM.probe( j );
          fire output RM.RECEIVE( m );
       od

          % -- Prepare the release signals
    fire internal LN.prepmessages(release);

     % -- Send
    for j :Nat where j < MPI_Size() do
       fire output LN.SEND(m);
    od
    follow SM.DELAY duration (MPI_Size() * 10);

    % -- Receive
    for j :Nat where j < MPI_Size() do
       m := nil();
       fire input  RM.probe( j );
          fire output RM.RECEIVE( m );
       od

          % -- Prepare the recovery signals
    fire internal LN.prepmessages(recovery);

     % -- Send
    for j :Nat where j < MPI_Size() do
       fire output LN.SEND(m);
    od
    follow SM.DELAY duration (MPI_Size() * 10);

    % -- Receive
    for j :Nat where j < MPI_Size() do
       m := nil();
       fire input  RM.probe( j );
          fire output RM.RECEIVE( m );
       od

      % -- try to enter the critical region
      fire internal LN.entercritical;

od % for on execution length
```

```
    %print LN.messagesreceived;
   %print LN.leastwaited;
   print "processID";
   print MPI_Rank();
   print "timesentered";
   print LN.timesentered;
   print "totalwaited";
   print LN.totalwaited;
   print "maxwaited";
   print LN.maxwaited ;
   print "messagesreceived";
   print LN.messagecounter;
od % end of schedule
```

B.3 Κώδικας για ένα εσωτερικό κόμβο Lelann (LelannNode.tioa)

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%
%%% .:Lelann Node
%
%%
%% Describes an internal Node as implemented for Lelann Algorithm
%% in distributed systems. It is fault tolerant in case of crash failures.
%
%%% .: ALGORITHM AUTOMATON :.
%%% u --> a frequency of advancing the trajectory
%%% MPI_Rank() --> the process id
%%% MPI_Size() --> the total of processes involved
%%% chancetodrop --> A percentage chance for this node to drop in each iteration.
for example 1 -> 1%.
%%% chancetoenter --> A percentage chance for this node to want to enter the
critical region
%%% in each iteration. for example 1 -> 1%.
%
%
%the msg field will be used as a means of defining the type of message sent.
% 1 - The Token message
% 2 - Acknowledge message
% 3 - Recovery message
%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%


automaton LelannNode(u, chancetodrop, chancetoenter,timeoutwait: DiscreteReal)
where u > 0
  signature
    output SEND( m:Null[mpi_message] )
    input  RECEIVE( m:Null[mpi_message] )
    internal prepmessages(type:Nat), init,entercritical,checkfail, checkenter,
checktimeout, updatetopology, timer

  states

    entered:                 AugmentedReal:=0;
    tosend:                         Seq[Null[mpi_message]] := { }; %the queue of
messages to be sent
    Nbrs:                    Seq[process_status] := { }; % Includes all the nodes of
the system
    index:                          Nat:=0;
    nextavailable:           Nat:=0;
    prevavailable:           Nat:=0;
    clock:                          AugmentedReal := 0; % The physical clock of the
automaton
    timerequested:           AugmentedReal := 0; % The time requested to enter in
the critical section.
    nextsend:              DiscreteReal:=0; % Time to stop the trajectory
    ProcessBegan:            Bool:=true; % Indicates a working process
    criticalsection:         Bool:=false; % Indicates entry into the critical region
    token:                          Bool:=false; %Indicates that the process has the
token
    wantstoenter:            Bool:=false; % Indicates the wish to enter the critical
region
```

```
    mustreply:                  Bool:=false; % Indicates that this process has to
send a reply message
    requestingack:        Bool:=false; % Indicates that the process is waiting
for an ack signal
    done:                       Bool:=false; % Indicates that the process is done
with the critical region
    donetimer:                  AugmentedReal:=0; % A timer to tell when the
process is done with the critical region. Decided randomly.
    recovered:                  Bool:=false; % Indicates that the process has
recovered from a crash
    messagecounter:       AugmentedReal:=0; % A counter to count how many
messages this node has sent
    recoverytime:         AugmentedReal:=0; % The time it will take to recover
after each failure. Decided randomly.
    recovercounter:       Int:=0; %counts how many times this process has
recovered from crash failures.
    timesentered:         Int:=0; %counts how many times this process has entered
the critical section.
    timewaited:                 AugmentedReal:=0; % Time waited from request to
entering for a single entrance into the critical section.
    maxwaited:                  AugmentedReal:=0; % Max time waited to enter.
    leastwaited:          AugmentedReal:=1000000; % Least time waited to enter.
    totalwaited:          AugmentedReal:=0; % Total time spent waiting to enter.



  transitions

     output SEND(m)
      pre
         tosend ~= { };
         m = head(tosend);
         ProcessBegan;
      eff
         tosend := tail(tosend);
         messagecounter:=messagecounter+1;


    input RECEIVE(m)
      eff
        if m ~= nil() then
            %Receive the token message
                  %also adds the sender in the reply queue in order to send him
the ack signal
            if(val(m).data=1) then
                  token:=true;
                        mustreply:=true;
                        prevavailable:=val(m).sender;
                  fi;
            %Receive the ack message
            if (val(m).data=2) then
                  requestingack:=false;
                  for n : Nat where n < len(Nbrs) do
                        if Nbrs[n].rank=val(m).sender then
                              Nbrs[n].online:=true;
                        fi
                  od;
                  nextavailable:=val(m).sender;
            fi;
                  %Receive the recovery signal
                  %Sets the node that sent it to active
            if (val(m).data=3) then
                        for n : Nat where n < len(Nbrs) do
                              if Nbrs[n].rank=val(m).sender then
```

```
                                        Nbrs[n].online:=true;
                                    fi
                                od;
                                nextavailable:=val(m).sender;
            fi;

         fi

      internal timer
       pre true;
       eff
          nextsend := nextsend + u;

      internal updatetopology
      locals
        dowhile:Bool:=true;
        temp:Nat:=0;
      pre ProcessBegan;
      eff
         %set the nextavailable value
         dowhile:=true;
         for n:Nat where n < len (Nbrs) do
               if Nbrs[mod(index+n+1,len(Nbrs))].online=true /\ dowhile then
                     nextavailable:=Nbrs[mod(index+n+1,len(Nbrs))].rank;
                     dowhile:=false;
               fi;
         od;

    internal prepmessages(type)
      pre len(tosend) = 0 /\ ProcessBegan;
      eff
                     if(type=1 /\ token /\ ~wantstoenter /\~criticalsection ) then
%token message
                     tosend := tosend |- embed([1,MPI_Rank(),nextavailable]);
                            requestingack:=true;
                            timerequested:=clock;
                            token:=false;
                            done:=false;
               fi;
               if(type=2 /\ mustreply) then   %ack signal
                     tosend := tosend |- embed([2,MPI_Rank(),prevavailable]);
                     mustreply:=false;
               fi;
               if(type=3 /\ recovered) then   %recovery signal
                     recovered:=false;
               tosend := tosend |- embed([3,MPI_Rank(),prevavailable]);
               fi;

    %set the index value
       %also initially set all the nodes to online
       %finally, creates the token at node 0
    internal init
       pre true;
      eff
         for n : Nat where n < MPI_Size() do
                     Nbrs:= Nbrs |- [n,true];
                     Nbrs[n].rank:=n;
                     Nbrs[n].online:=true;
                     if n=MPI_Rank() then index:=n; fi;
         od
         if MPI_Rank()=0 then token:=true; fi;

       %randomly checks if the process will fail
      %the chance is given by the parameter chancetofail
```

```
      %also decides a random recovery time between 10 and 40 rounds
      %when the recovery time is there, the process will be enabled again
      internal checkfail
      locals
        i:Int:=0;
        j:Int:=0;
       pre chancetodrop>0;
      eff
        if(ProcessBegan) then
             i:=choose i;
             i:= abs(i);
             i:=mod(i,100);
             if i<=chancetodrop then
                   ProcessBegan:=false;
                   j:= choose j;
                   j:=abs(j);
                   j:=mod(j,10)+5;
                   recoverytime:=clock + j*u;
                   token:=false;
                   wantstoenter:=false;
                   criticalsection:=false;
             fi
        else
             if clock>=recoverytime then
                   ProcessBegan:=true;
                   recovered:=true;
                   recovercounter:=recovercounter+1;
             fi
        fi

      %randomly checks if the process will want to enter the critical region
      %the chance is given by the parameter chancetoenter
      internal checkenter
      locals
        i:Int:=0;
       pre ProcessBegan /\ ~wantstoenter /\ ~criticalsection /\ ~done;
      eff
        i:=choose i;
        i:= abs(i);
        i:=mod(i,100);
        if i<chancetoenter then
             wantstoenter:=true;
             timerequested := clock;
             done:=false;
        fi


      %checks if the next node timed out
      %is triggered 'timeoutwait' rounds after the process has requested a reply
      %as given by the parameter
      internal checktimeout
        pre ProcessBegan /\ requestingack /\ clock >= timerequested + timeoutwait
* u;
        eff
             for i : Nat where i < len(Nbrs) do
                   if Nbrs[i].rank=nextavailable then Nbrs[i].online:=false; fi;

             od
             token:=true;
             wantstoenter:=false;
             done:=true;


  %enters the critical section only if the token is in the process's queue
```

```
  %and the process does want to enter
  %also exits the critical region when the time has come to do so
  internal entercritical
  locals
  i:Int:=0;
  pre
       ProcessBegan;
  eff
       if(token /\ wantstoenter /\ ~criticalsection) then
             criticalsection:=true;
             wantstoenter:=false;
             done:=false;
             timesentered:=timesentered +1;
             timewaited:=clock - timerequested;
             totalwaited:= totalwaited + timewaited;
             if timewaited < leastwaited then leastwaited:=timewaited;fi;
             if timewaited > maxwaited then maxwaited:=timewaited;fi;
        i:=choose i;
        i:= abs(i);
        i:=mod(i,10)+10;
             donetimer:=clock+i*u;
    fi;
    if criticalsection /\ clock >= donetimer then criticalsection:=false;
done:=true; fi;


  trajectories
    trajdef T
    stop when clock>nextsend;
      evolve d(clock) = 1;
```

B.4 Κώδικας για ένα ολοκληρωμένο κόμβο Lelann (LelannSystem.tioa)

```
%%% .: VOCABS :.
include "MPIVocabs.tioa"

imports mpi_message_voc, mpi_request_voc, mpi_status_voc, mpi_voc,
alg_specific_voc

%%% .: MPI AUTOMATA :.
include "ReceiveMediator.tioa", "SendMediator.tioa"

%%% .: ALG AUTOMATON :.
include "LelannNode.tioa"

% -- eclipse running instructions
% program arguments:-ea 100 2 5 20 200
% for cluster we also need to add this parameter: -dev niodev
% VM arguments: -jar ${MPJ_HOME}/lib/starter.jar -np 5

automaton LelannSystem(u,chancetodrop,chancetoenter,timeoutwait: DiscreteReal,
runs:Nat)
  components
    LN : LelannNode(u,chancetodrop,chancetoenter,timeoutwait);
    SM : SendMediator;
       RM : ReceiveMediator;

schedule
states
   m : Null[mpi_message] := nil();
   request   :Nat:=1;
   ack       :Nat:=2;
   recovery  :Nat:=3;

do

   fire internal LN.init;
   % -- From here starts the main loop of the execution
   for i : Nat where i < runs do
      follow LN.T duration \infty;

        % -- Fix the timer of this run
      fire internal LN.timer;

        % -- Check if the process will randomly fail
      fire internal LN.checkfail;

      % -- Check if the process randomly wants to enter the critical region
      fire internal LN.checkenter;

      % -- Check if some nodes timed out
      fire internal LN.checktimeout;

        % -- update the topology of the network
        fire internal LN.updatetopology;

       % -- Prepare the request signals
      fire internal LN.prepmessages(request);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
```

```
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
         od

       % -- Prepare the ack signals
      fire internal LN.prepmessages(ack);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
         od

       % -- Prepare the recovery signals
      fire internal LN.prepmessages(recovery);

       % -- Send
      for j :Nat where j < MPI_Size() do
         fire output LN.SEND(m);
      od
      follow SM.DELAY duration (MPI_Size() * 10);

      % -- Receive
      for j :Nat where j < MPI_Size() do
         m := nil();
         fire input  RM.probe( j );
            fire output RM.RECEIVE( m );
         od

        % -- try to enter the critical region
        fire internal LN.entercritical;



   od % for on execution length


    %print LN.messagesreceived;
   %print LN.leastwaited;
   print "processID";
   print MPI_Rank();
   print "timesentered";
   print LN.timesentered;
   print "totalwaited";
   print LN.totalwaited;
   print "maxwaited";
   print LN.maxwaited ;
   print "messagessent";
   print LN.messagecounter;
od % end of schedule
```